Universidad Tecnológica de Pereira

Mision TIC 2022

El futuro digital es de todos

MinTIC

Unidad 6

18 - Patrones de Diseño Gang of Four

Hechos QUE CONECTAN

¿Que es un patrón de Diseño?

- Los patrones de diseño son soluciones habituales a problemas que ocurren con frecuencia en el diseño de software.
- Son como planos prefabricados que se pueden personalizar para resolver un problema de diseño recurrente en tu código.
- No se puede elegir un patrón y copiarlo en el programa como si se tratara de funciones o bibliotecas ya preparadas.
- El patrón no es una porción específica de código, sino un concepto general para resolver un problema particular.
- Puedes seguir los detalles del patrón e implementar una solución que encaje con las realidades de tu propio programa.

Historia de los patrones

- El concepto de los patrones fue descrito por Christopher Alexander en **El lenguaje de patrones**. El libro habla de un “lenguaje” para diseñar el entorno urbano. Las unidades de este lenguaje son los patrones. Pueden describir lo altas que tienen que ser las ventanas, cuántos niveles debe tener un edificio, cuán grandes deben ser las zonas verdes de un barrio, etcétera.
- La idea fue recogida por cuatro autores: Erich Gamma, John Vlissides, Ralph Johnson y Richard Helm. En 1995, publicaron **Patrones de diseño**, en el que aplicaron el concepto de los patrones de diseño a la programación.
- El libro presentaba 23 patrones que resolvían varios problemas del diseño orientado a objetos y se convirtió en un éxito de ventas con rapidez. Al tener un título tan largo en inglés, la gente empezó a llamarlo “**El libro de la ‘Gang of Four’ (banda de los cuatro)**”, lo que pronto se abrevió a “**El libro GoF**”.

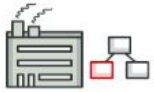
¿Por qué debería aprender sobre patrones?

- La realidad es que ***podrías trabajar durante años como programador sin conocer un solo patrón***. Mucha gente lo hace. Incluso en ese caso, podrías estar implementando patrones sin saberlo.
- Los patrones de diseño son un juego de herramientas de soluciones comprobadas a problemas habituales en el diseño de software. Incluso aunque nunca te encuentres con estos problemas, conocer los patrones sigue siendo de utilidad, porque te enseña a resolver todo tipo de problemas utilizando principios del diseño orientado a objetos.
- Los patrones de diseño definen un lenguaje común que puedes utilizar con tus compañeros de equipo para comunicaros de forma más eficiente. Podrías decir: "***Oh, utiliza un singleton para eso***", y todos entenderían la idea de tu sugerencia. No habría necesidad de explicar qué es un singleton si conocen el patrón y su nombre.

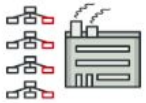
Clasificación de patrones

- Los patrones de diseño varían en su complejidad, nivel de detalle y escala de aplicabilidad al sistema completo que se diseña.
- Los patrones más básicos y de más bajo nivel suelen llamarse ***idioms***. Normalmente se aplican a un único lenguaje de programación.
- Los patrones más universales y de más alto nivel son los ***patrones de arquitectura***. Se pueden implementar estos patrones en cualquier lenguaje.
- Todos los patrones pueden clasificarse por su propósito:
  - Los **patrones creacionales** proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización de código existente.
  - Los **patrones estructurales** explican cómo ensamblar objetos y clases en estructuras más grandes a la vez que se mantiene la flexibilidad y eficiencia de la estructura.
  - Los **patrones de comportamiento** se encargan de una comunicación efectiva y la asignación de responsabilidades entre objetos.

Patrones GoF

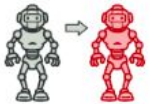


Factory Method



Abstract Factory

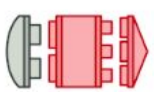
Builder



Prototype

Singleton

Creacionales



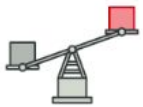
Adapter

Bridge

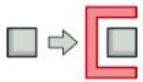
Composite

Decorator

Facade



Flyweight



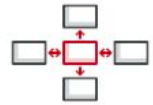
Proxy

Estructurales

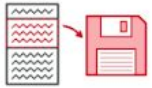
Chain of Responsibility

Command

Iterator

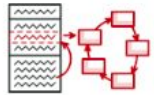


Mediator

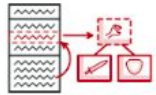


Memento

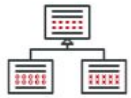
Observer



State



Strategy



Template Method

Visitor

De comportamiento

¿En qué consiste un patrón?

- La mayoría de los patrones se describe con mucha formalidad para que la gente pueda reproducirlos en muchos contextos.
  - El **propósito** del patrón explica brevemente el problema y la solución.
  - La **motivación** explica en más detalle el problema y la solución que brinda el patrón.
  - La **estructura** de las clases muestra cada una de las partes del patrón y el modo en que se relacionan.
  - El **ejemplo de código** en uno de los lenguajes de programación populares facilita la asimilación de la idea que se esconde tras el patrón.

Universidad Tecnológica de Pereira

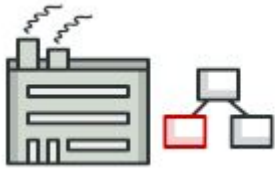
Mision TIC 2022

El futuro digital es de todos

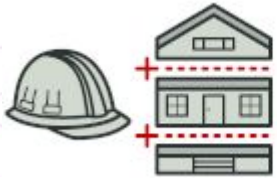
MinTIC

Patrones creacionales

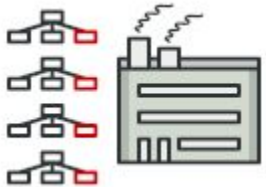
Hechos QUE CONECTAN



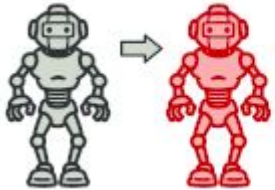
Factory Method



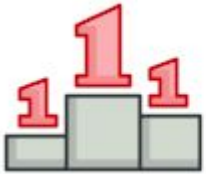
Builder



Abstract Factory



Prototype



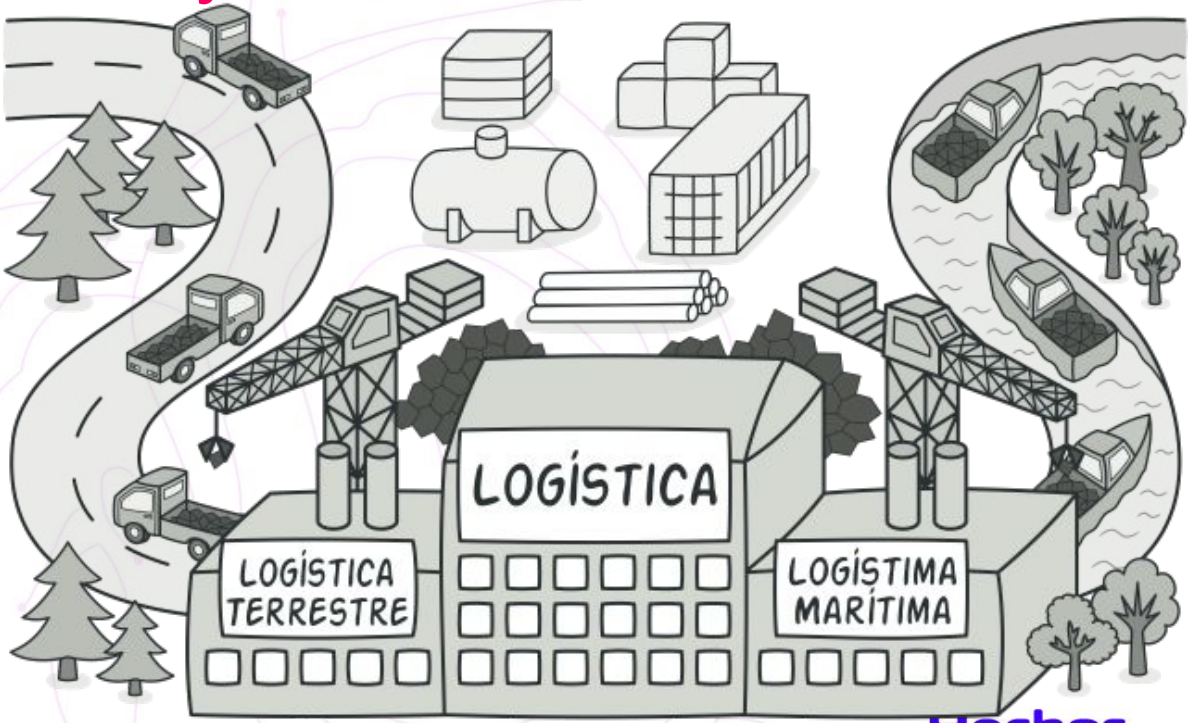
Singleton

Patrones crecionales

Factory Method

Propósito

Es un patrón que proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.



Factory Method

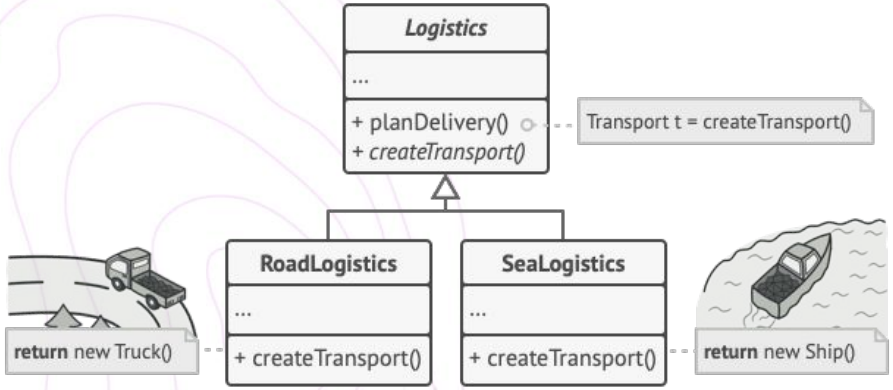
Problema

- Imagina que estás creando una aplicación de gestión logística. La primera versión de tu aplicación sólo es capaz de manejar el transporte en camión, por lo que la mayor parte de tu código se encuentra dentro de la clase **Camión**.
- Al cabo de un tiempo, tu aplicación se vuelve bastante popular. Cada día recibes decenas de peticiones de empresas de transporte marítimo para que incorpores la logística por mar a la aplicación.
- ¿Qué pasa con el código?La mayor parte de tu código está acoplado a la clase **Camión**. Para añadir barcos a la aplicación habría que hacer cambios en toda la base del código. Además, si más tarde decides añadir otro tipo de transporte a la aplicación, probablemente tendrás que volver a hacer todos estos cambios.

Factory Method

Solución

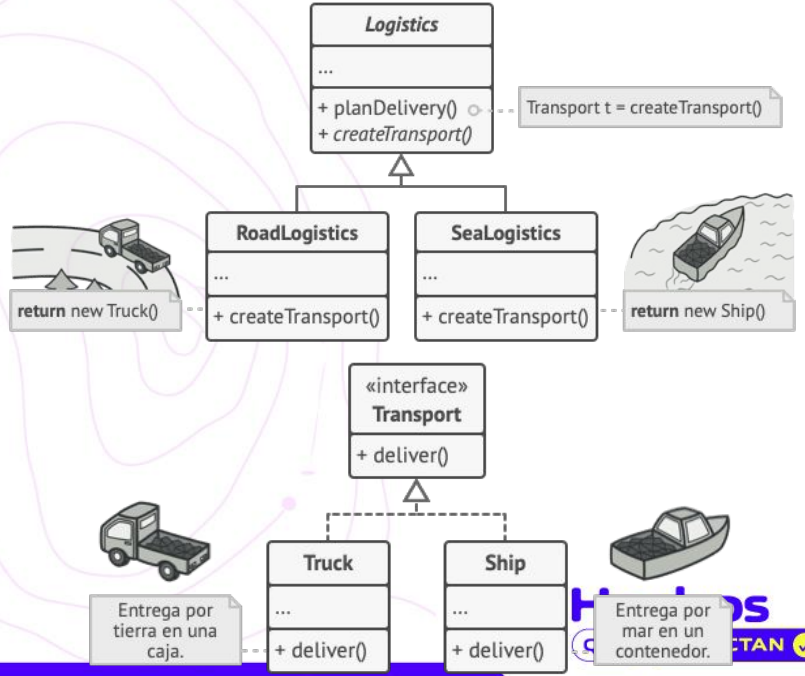
- El patrón sugiere que, en lugar de llamar al operador ***new*** para construir objetos directamente, se invoque a un **método fábrica** especial. No te preocupes: los objetos se siguen creando a través del operador new, pero se invocan desde el método fábrica. Los objetos devueltos por el método fábrica a menudo se denominan *productos*.



Factory Method

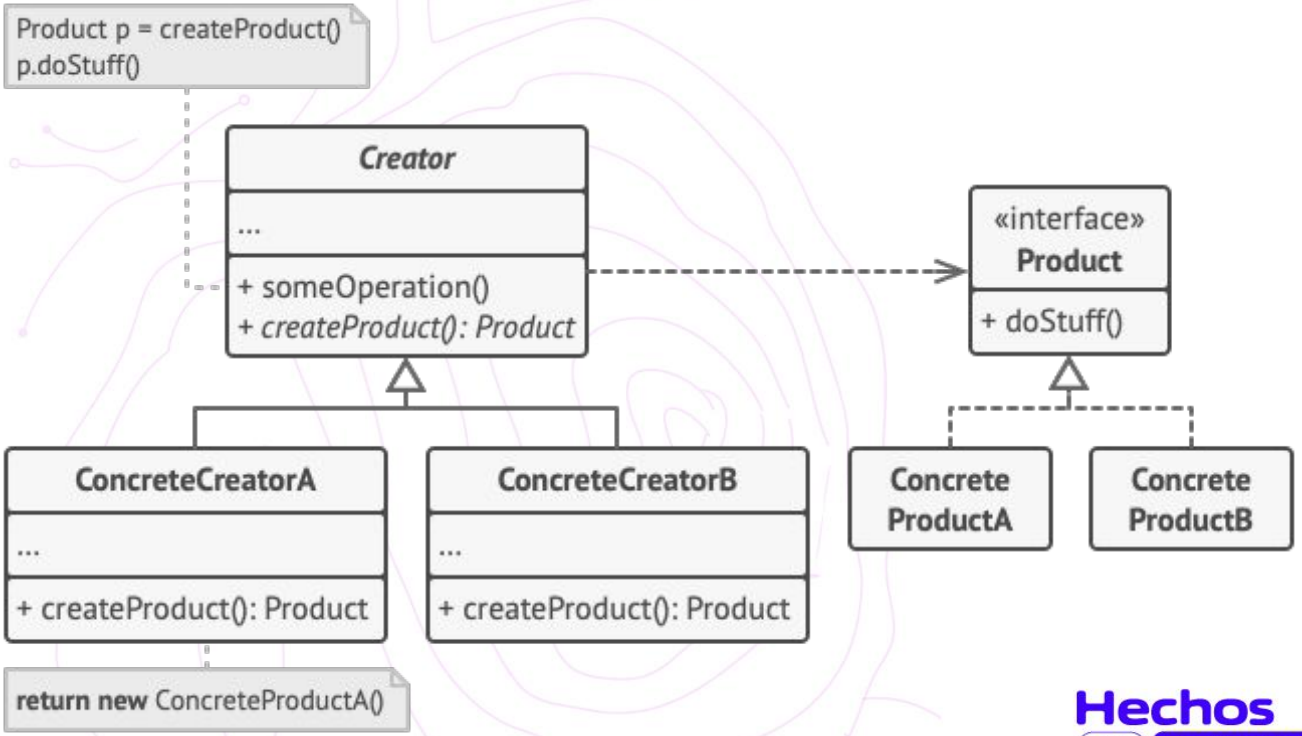
Solución

- El patrón sugiere que, en lugar de llamar al operador ***new*** para construir objetos directamente, se invoque a un **método fábrica** especial. No te preocupes: los objetos se siguen creando a través del operador new, pero se invocan desde el método fábrica. Los objetos devueltos por el método fábrica a menudo se denominan *productos*.
- Las subclases sólo pueden devolver *productos* de distintos tipos si dichos productos tienen una clase base o interfaz común. Además, el método fábrica en la clase base debe tener su tipo de retorno declarado como dicha interfaz.

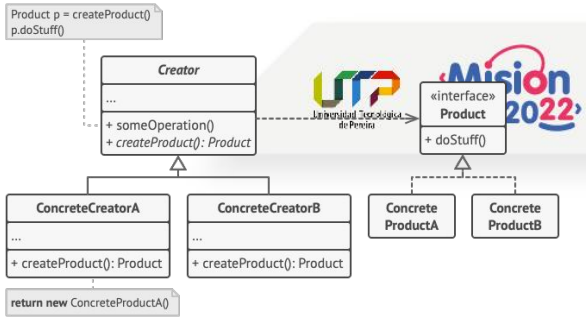


Factory Method

Solución General



Factory Method



Ventajas

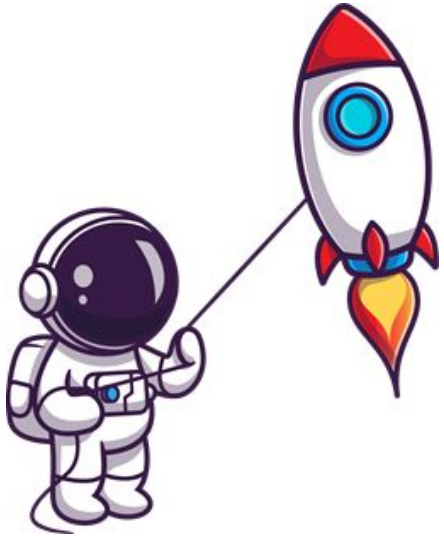
- Evitas un acoplamiento fuerte entre el creador y los productos concretos.
- ***Principio de responsabilidad única***. Puedes mover el código de creación de producto a un lugar del programa, haciendo que el código sea más fácil de mantener.
- ***Principio de abierto/cerrado***. Puedes incorporar nuevos tipos de productos en el programa sin descomponer el código cliente existente.

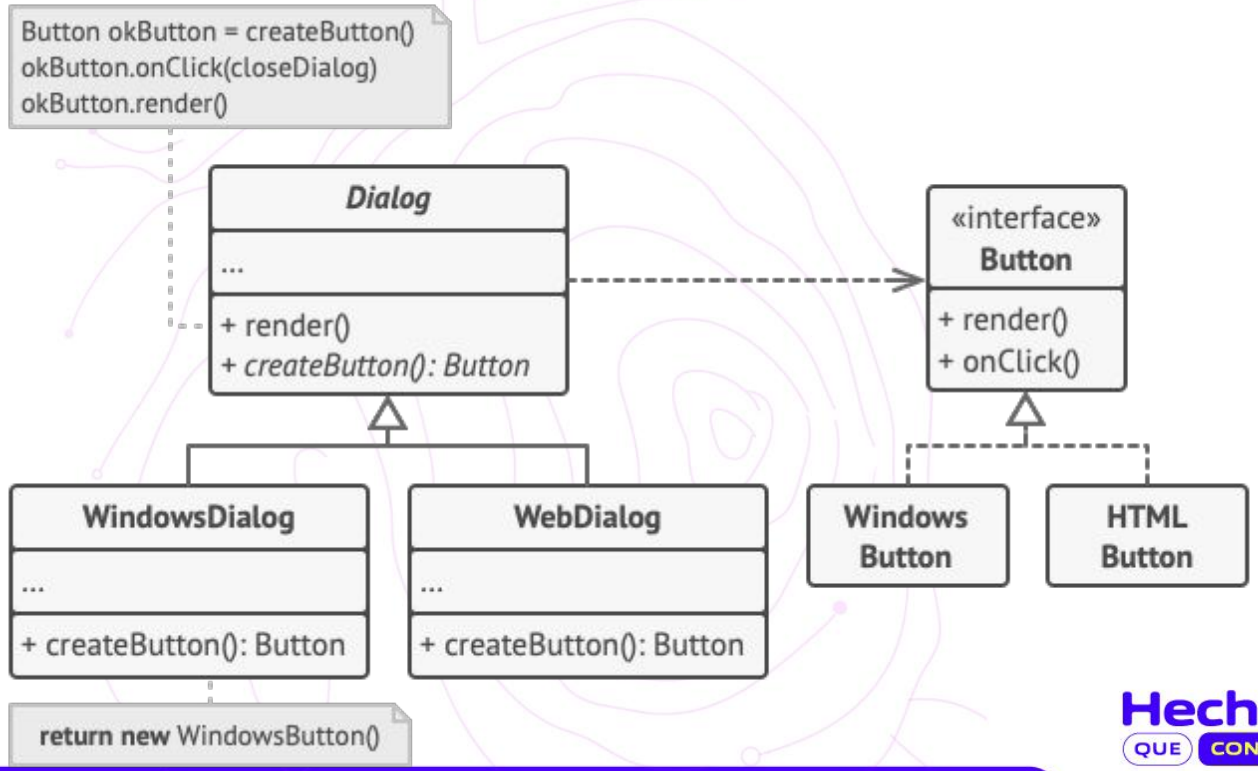
Desventajas

- Puede ser que el código se complique, ya que debes incorporar una multitud de nuevas subclases para implementar el patrón. La situación ideal sería introducir en una jerarquía existente de clases creadoras.

Factory Method

Ejemplo

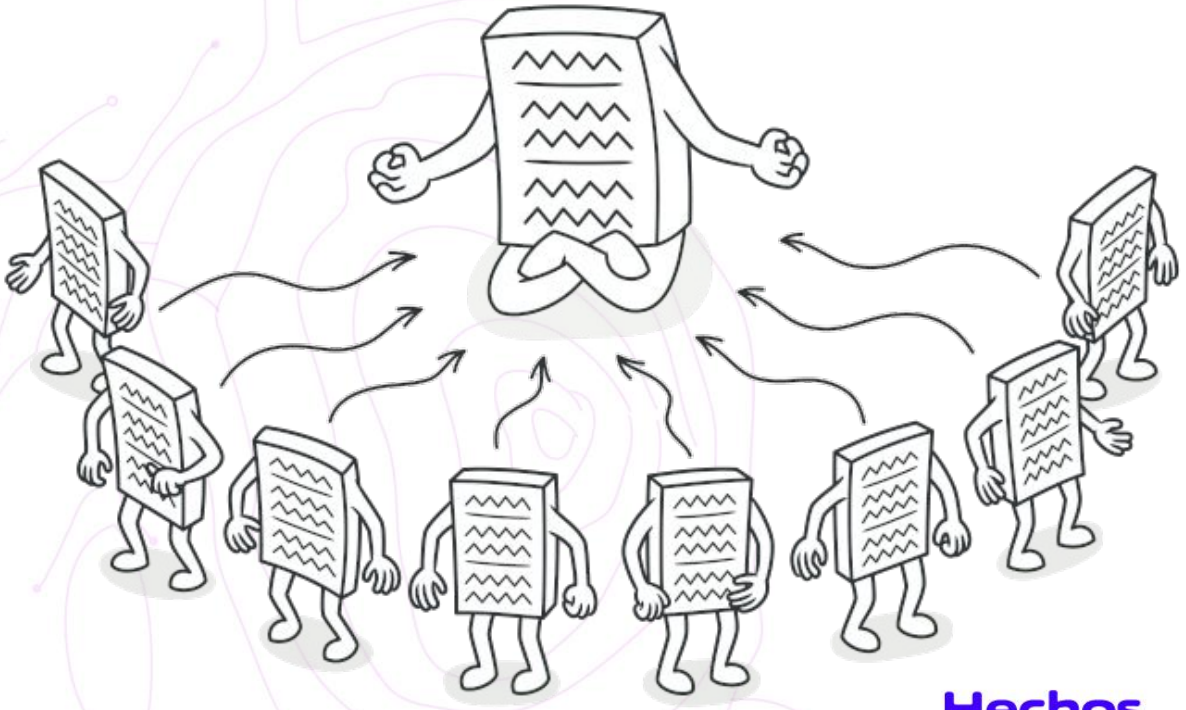




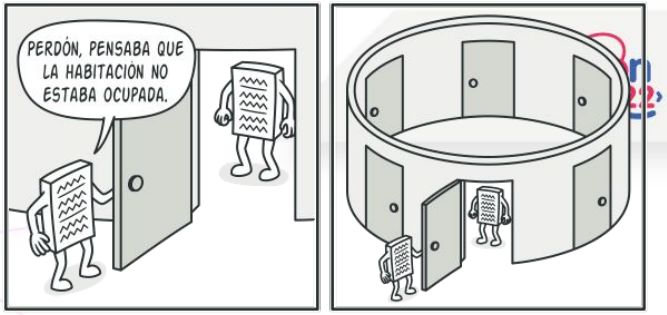
Singleton

Propósito

Es un patrón que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.



Singleton

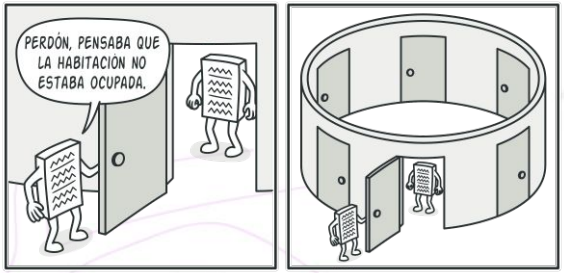


Problema

El patrón Singleton resuelve dos problemas al mismo tiempo, vulnerando el Principio de responsabilidad única:

1. **Garantizar que una clase tenga una única instancia**. ¿Por qué querría alguien controlar cuántas instancias tiene una clase? El motivo más habitual es controlar el acceso a algún recurso compartido, por ejemplo, una base de datos o un archivo.
2. **Proporcionar un punto de acceso global a dicha instancia**. ¿Recuerdas esas variables globales que utilizaste para almacenar objetos esenciales? Aunque son muy útiles, también son poco seguras, ya que cualquier código podría sobrescribir el contenido de esas variables y descomponer la aplicación.

Singleton



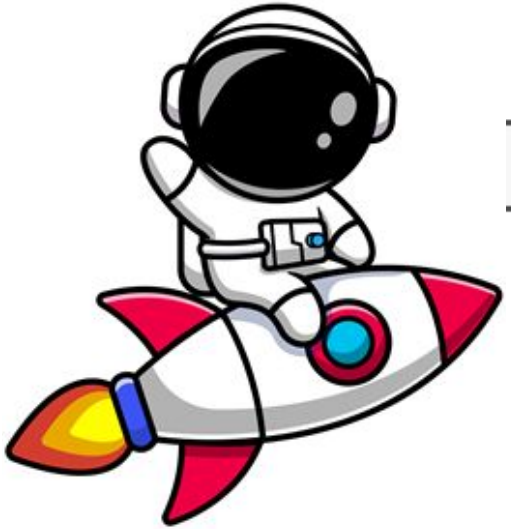
Solución

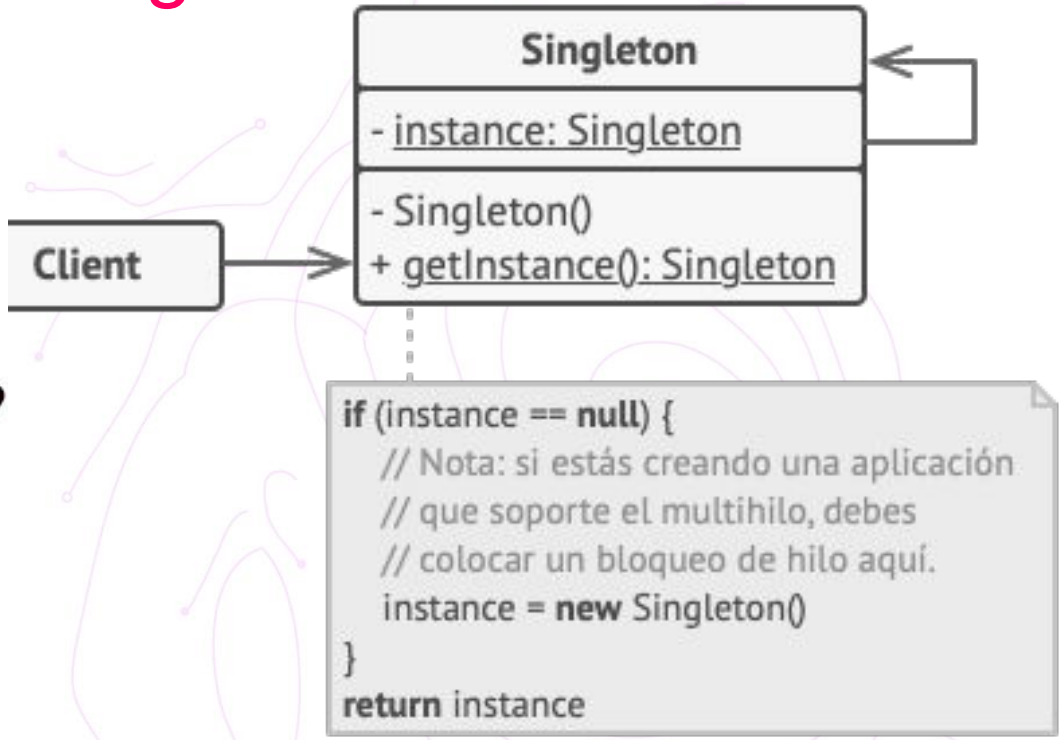
- Hacer privado el constructor por defecto para evitar que otros objetos utilicen el operador **new** con la clase Singleton.
- Crear un método de creación estático que actúe como constructor. Tras bambalinas, este método invoca al constructor privado para crear un objeto y lo guarda en un campo estático. Las siguientes llamadas a este método devuelve el objeto almacenado en caché.

Si tu código tiene acceso a la clase Singleton, podrá invocar su método estático. De esta manera, cada vez que se invoque este método, siempre se devolverá el mismo objeto.

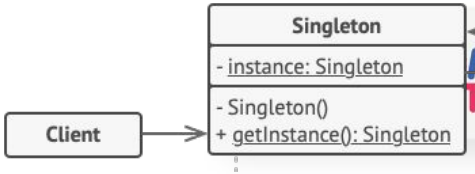
Singleton

Solución General





Singleton



Ventajas

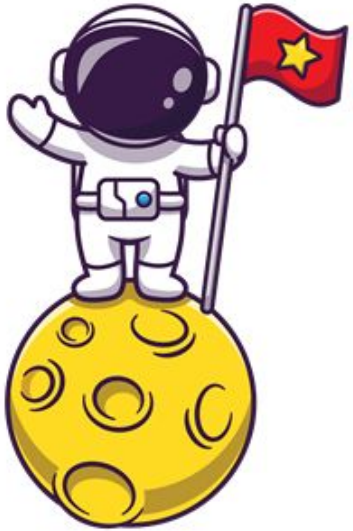
- Puedes tener la certeza de que una clase tiene una única instancia.
- Obtienes un punto de acceso global a dicha instancia.
- El objeto Singleton solo se inicializa cuando se requiere por primera vez.

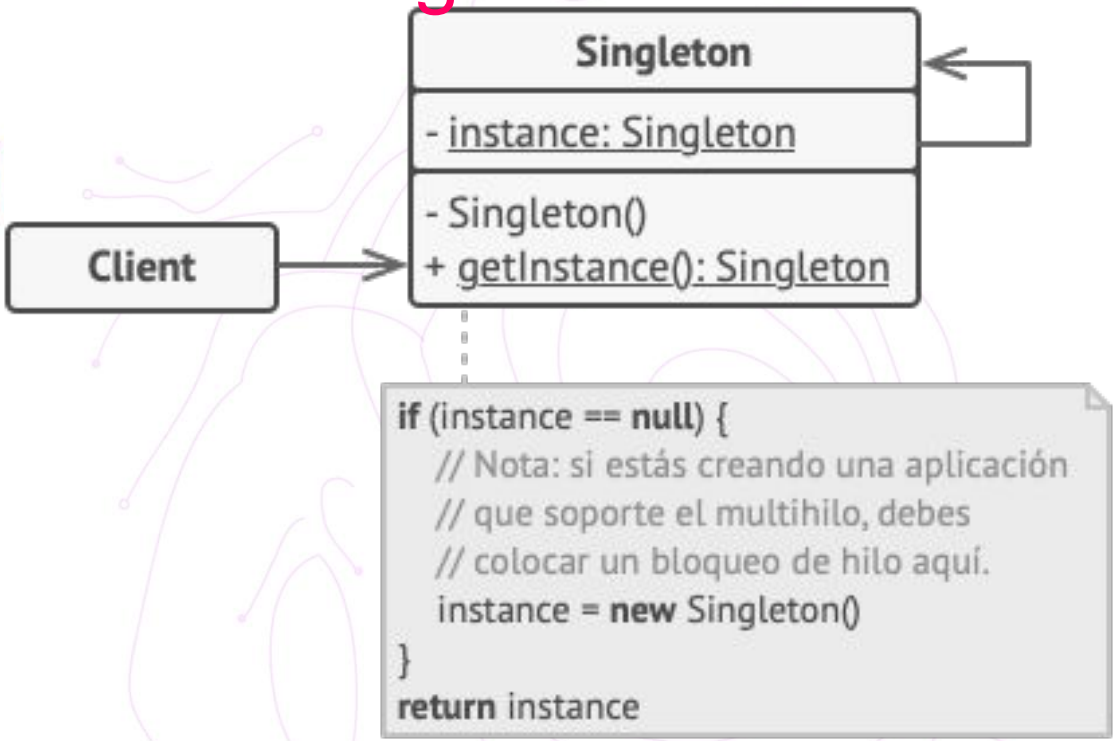
Desventajas

- Vulnera el *Principio de responsabilidad única*.
- Puede enmascarar un mal diseño, cuando los componentes del programa saben demasiado los unos sobre los otros.
- El patrón requiere de un tratamiento especial en un entorno con múltiples hilos de ejecución, para que varios hilos no creen un objeto Singleton varias veces.
- Puede resultar complicado realizar la prueba unitaria del código cliente del Singleton.

Singleton

Ejemplo

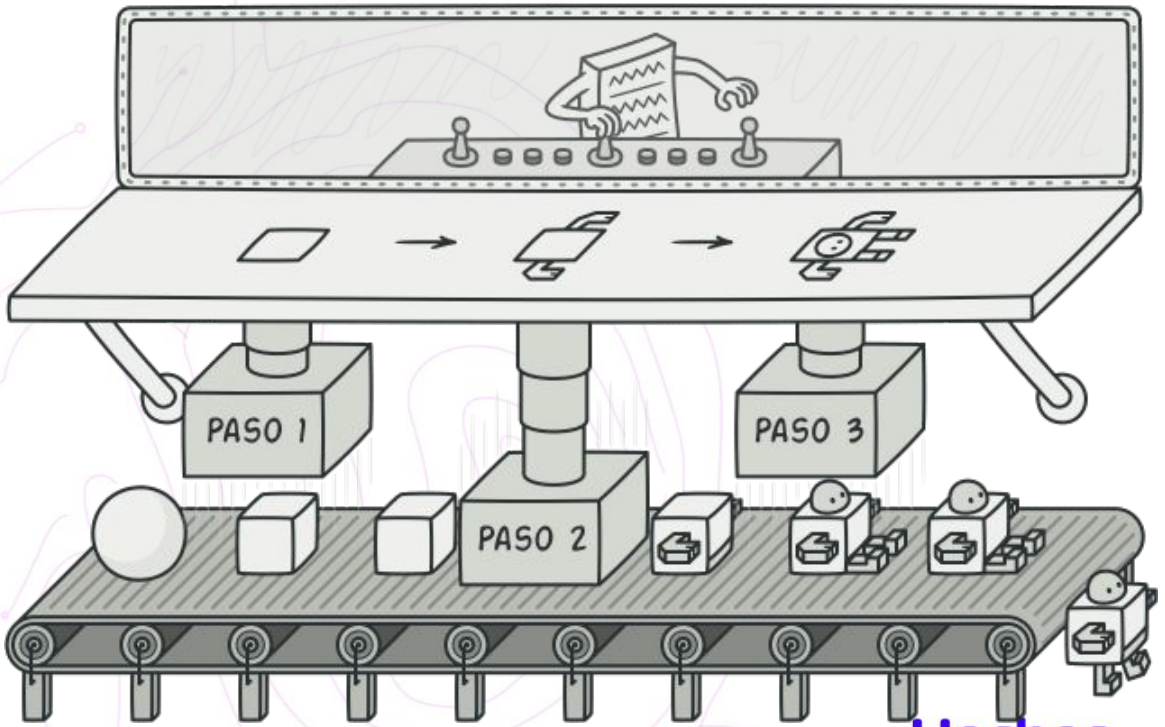




Builder

Propósito

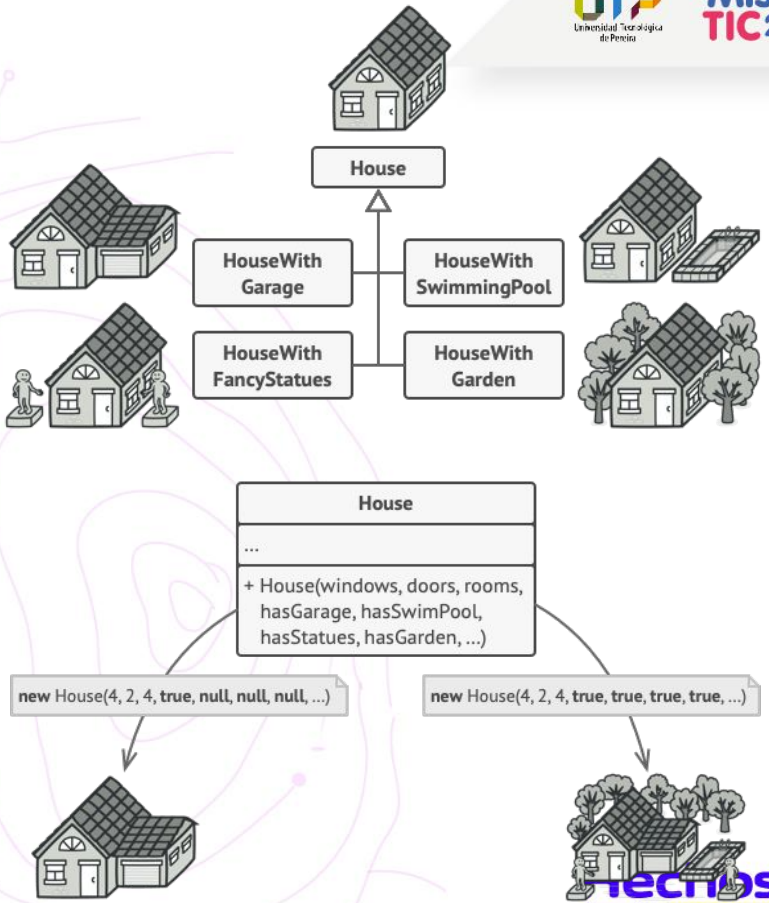
Es un patrón que nos permite construir objetos complejos paso a paso. El patrón nos permite producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción.



Builder

Problema

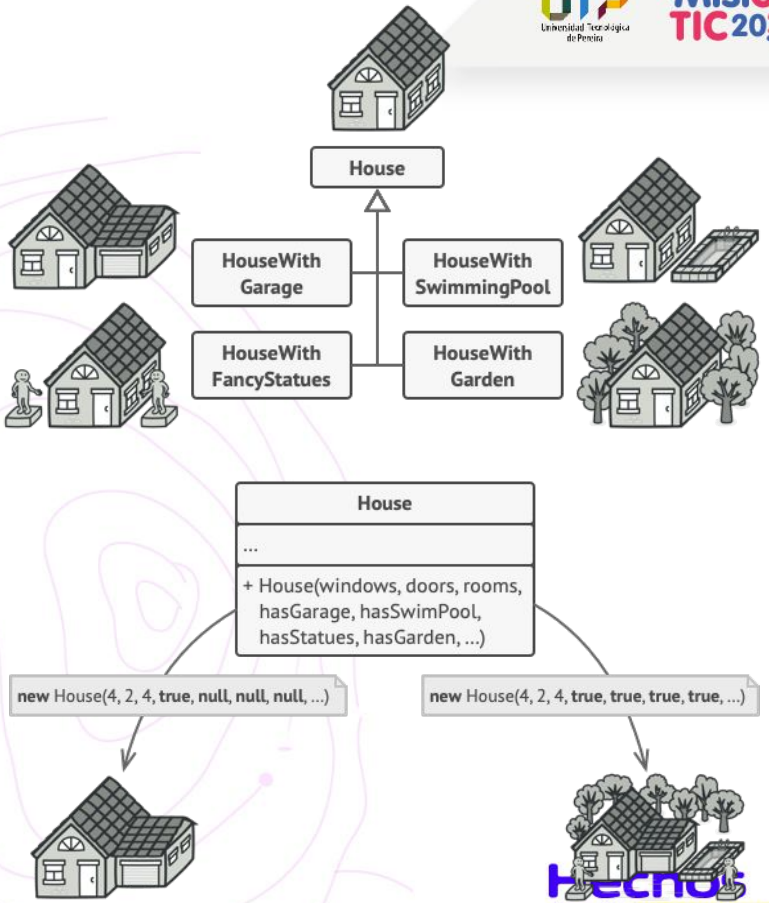
- Imagina un objeto complejo que requiere una inicialización laboriosa, paso a paso, de muchos campos y objetos anidados.
- Pensemos en cómo crear un objeto **Casa**. Para construir una casa sencilla, debemos construir cuatro paredes y un piso, así como instalar una puerta, colocar un par de ventanas y ponerle un tejado. Pero ¿qué pasa si quieres una casa más grande y luminosa, con un jardín y otros extras (como sistema de calefacción, instalación de fontanería y cableado eléctrico)?



Builder

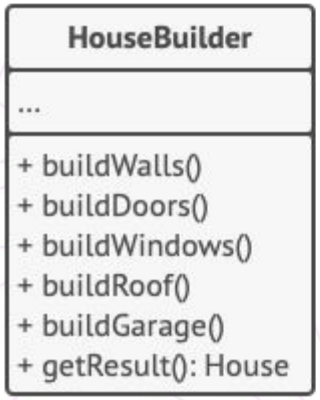
Problema

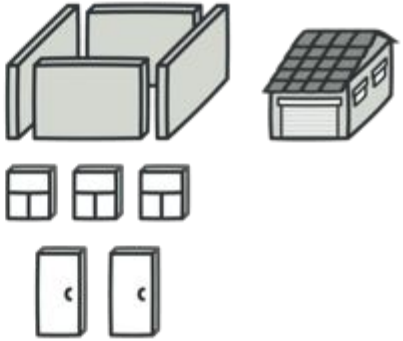
- La solución más sencilla es extender la clase base Casa y crear un grupo de subclases que cubran todas las combinaciones posibles de los parámetros. Pero, en cualquier caso, acabarás con una cantidad considerable de subclases.
- Existe otra posibilidad que no implica generar subclases. Puedes crear un enorme constructor dentro de la clase base Casa con todos los parámetros posibles para controlar el objeto casa. Aunque es cierto que esta solución elimina la necesidad de las subclases, genera otro problema.



Builder

Solución





- El patrón sugiere que saques el código de construcción del objeto de su propia clase y lo coloques dentro de objetos independientes llamados constructores.
- Lo importante es que no necesitas invocar todos los pasos. Puedes invocar sólo aquellos que sean necesarios para producir una configuración particular de un objeto.
- Puede ser que algunos pasos de la construcción necesiten una implementación diferente cuando tengamos que construir distintas representaciones del producto. Por ejemplo, las paredes de una cabaña pueden ser de madera, pero las paredes de un castillo tienen que ser de piedra.

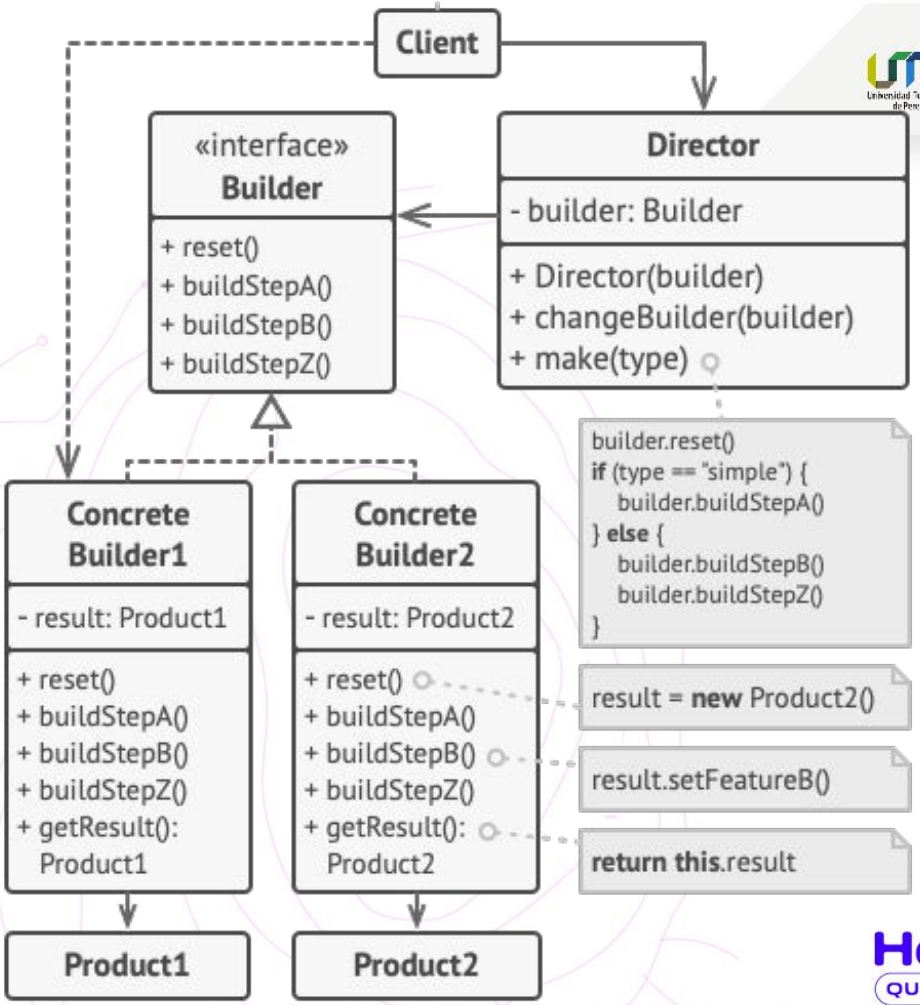
Builder

Solución

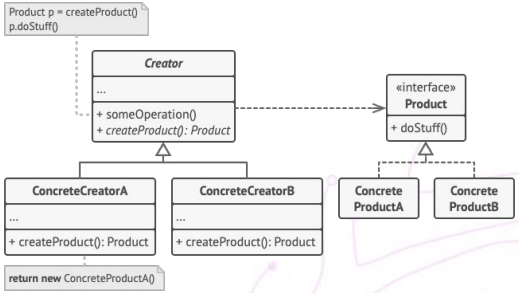
- Podemos crear varias clases constructoras distintas que implementen la misma serie de pasos de construcción, pero de forma diferente.
- Extraer una serie de llamadas a los pasos del constructor que utilizas para construir un producto y ponerlas en una clase independiente llamada directora.
- La clase directora define el orden en el que se deben ejecutar los pasos de construcción, mientras que el constructor proporciona la implementación de dichos pasos.

Builder

Solución General



Builder

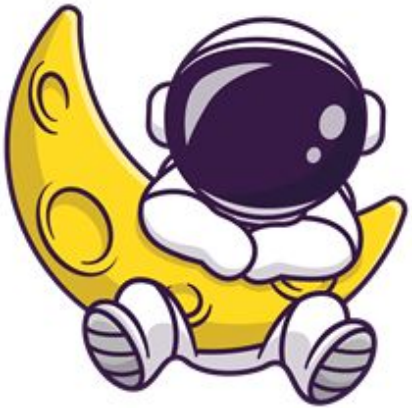


Ventajas

- Puedes construir objetos paso a paso, aplazar pasos de la construcción o ejecutar pasos de forma recursiva.
- Puedes reutilizar el mismo código de construcción al construir varias representaciones de productos.
- ***Principio de responsabilidad única***. Puedes aislar un código de construcción complejo de la lógica de negocio del producto.

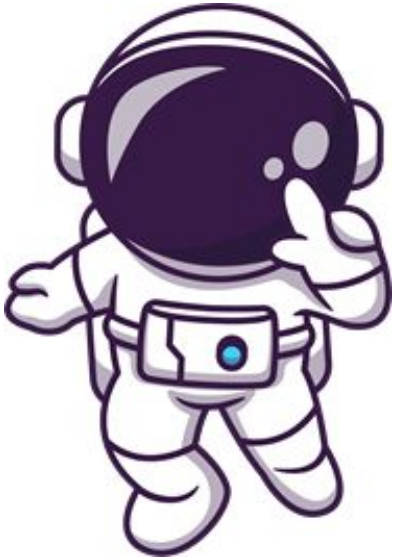
Desventajas

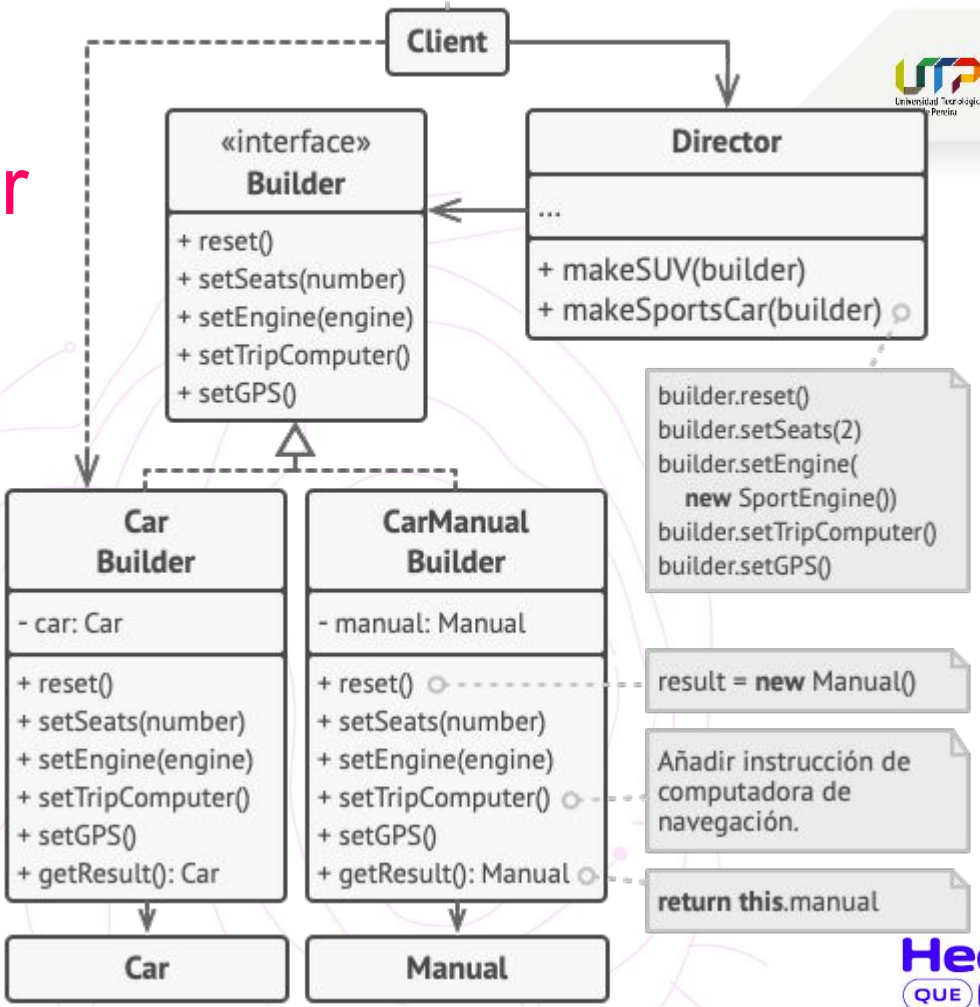
- La complejidad general del código aumenta, ya que el patrón exige la creación de varias clases nuevas.



Builder

Ejemplo

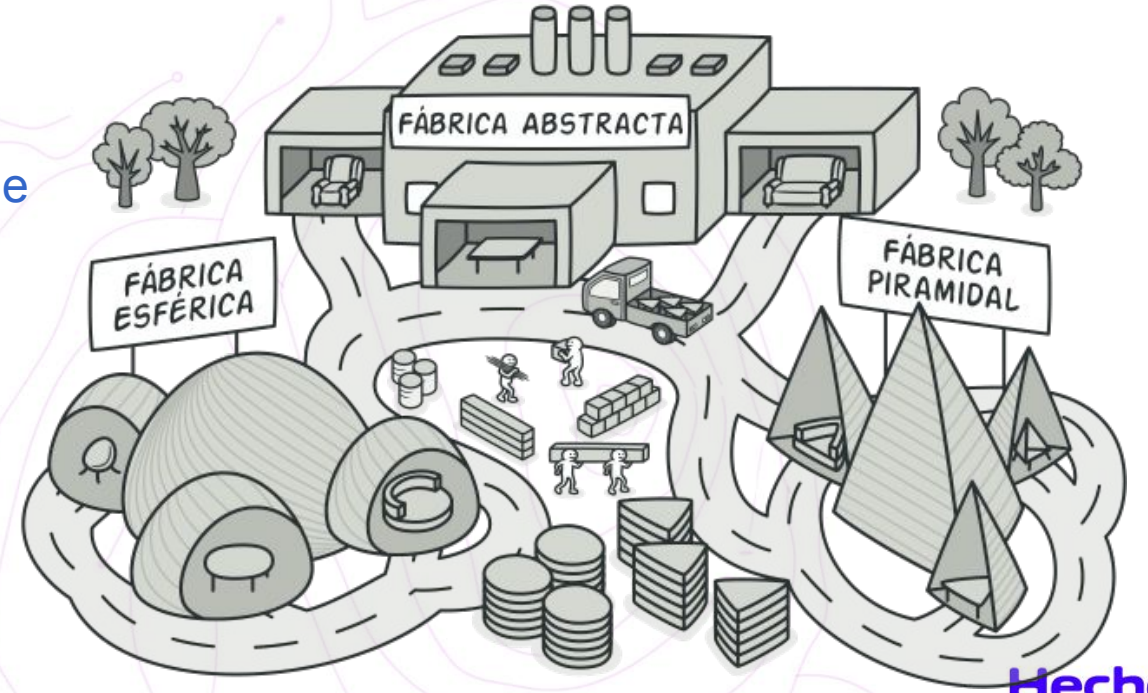




Abstract Factory

Propósito

Es un patrón que nos permite producir familias de objetos relacionados sin especificar sus clases concretas.



Abstract Factory

Problema

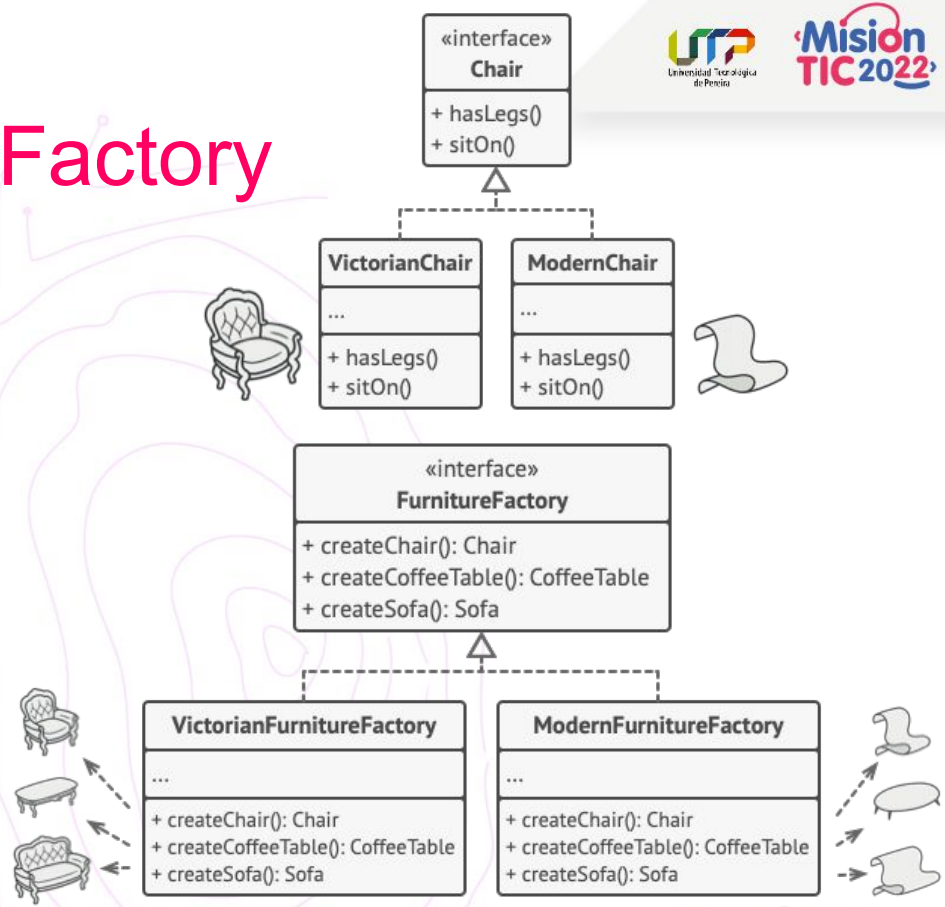
Imagina que estás creando un simulador de tienda de muebles. Tu código está compuesto por clases que representan lo siguiente:

1. Una familia de productos relacionados, digamos: Silla + Sofá + Mesilla.
2. Algunas variantes de esta familia. Por ejemplo, los productos Silla + Sofá + Mesilla están disponibles en estas variantes: Moderna, Victoriana, ArtDecó.

Abtract Factory

Solución

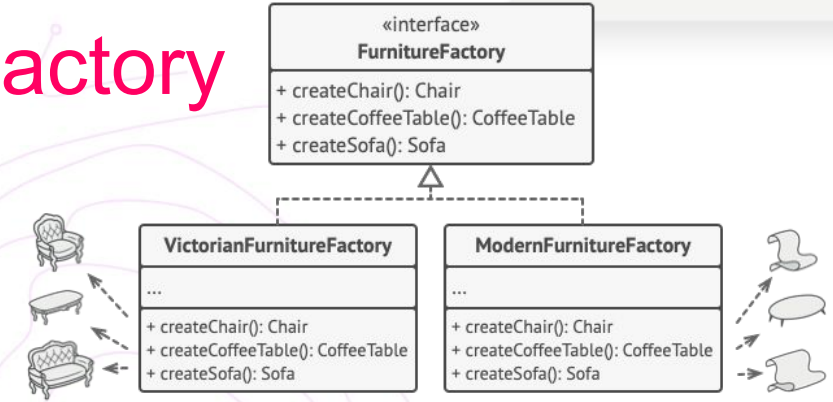
- El patrón sugiere que declaremos de forma explícita interfaces para cada producto diferente de la familia de productos (por ejemplo, silla, sofá o mesilla). Después podemos hacer que todas las variantes de los productos sigan esas interfaces.
- El siguiente paso consiste en declarar la Fábrica abstracta: una interfaz con una lista de métodos de creación para todos los productos que son parte de la familia de productos (por ejemplo, crearSilla, crearSofá y crearMesilla).



Abtract Factory

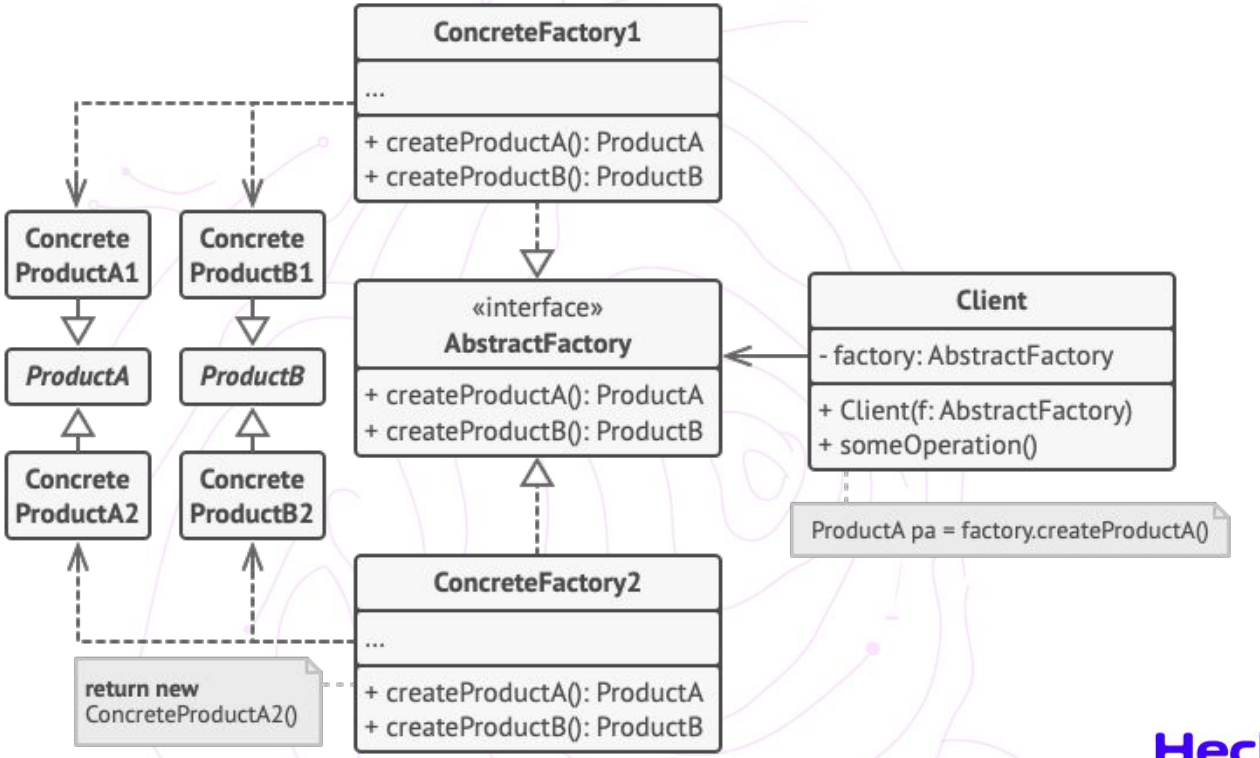
Solución

- Para cada variante de una familia de productos, creamos una clase de fábrica independiente basada en la interfaz FábricaAbstracta.
- Una fábrica es una clase que devuelve productos de un tipo particular. Por ejemplo, la ***FábricadeMueblesModernos*** sólo puede crear objetos de *SillaModerna*, *SofáModerno* y *MesillaModerna*.
- El cliente tiene que funcionar con fábricas y productos a través de sus respectivas interfaces abstractas.

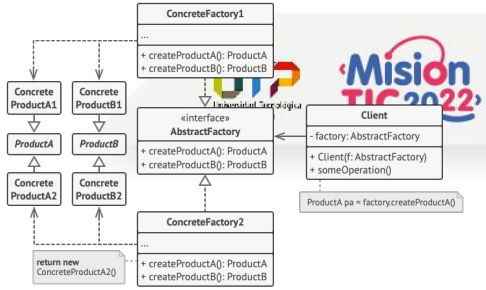


Abstract Factory

Solución General



Abstract Factory



Ventajas

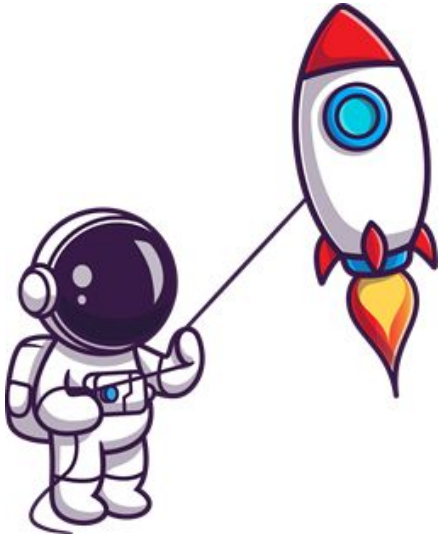
- Puedes tener la certeza de que los productos que obtienes de una fábrica son compatibles entre sí.
- Evitas un acoplamiento fuerte entre productos concretos y el código cliente.
- ***Principio de responsabilidad única***. Puedes mover el código de creación de productos a un solo lugar, haciendo que el código sea más fácil de mantener.
- ***Principio de abierto/cerrado***. Puedes introducir nuevas variantes de productos sin descomponer el código cliente existente.

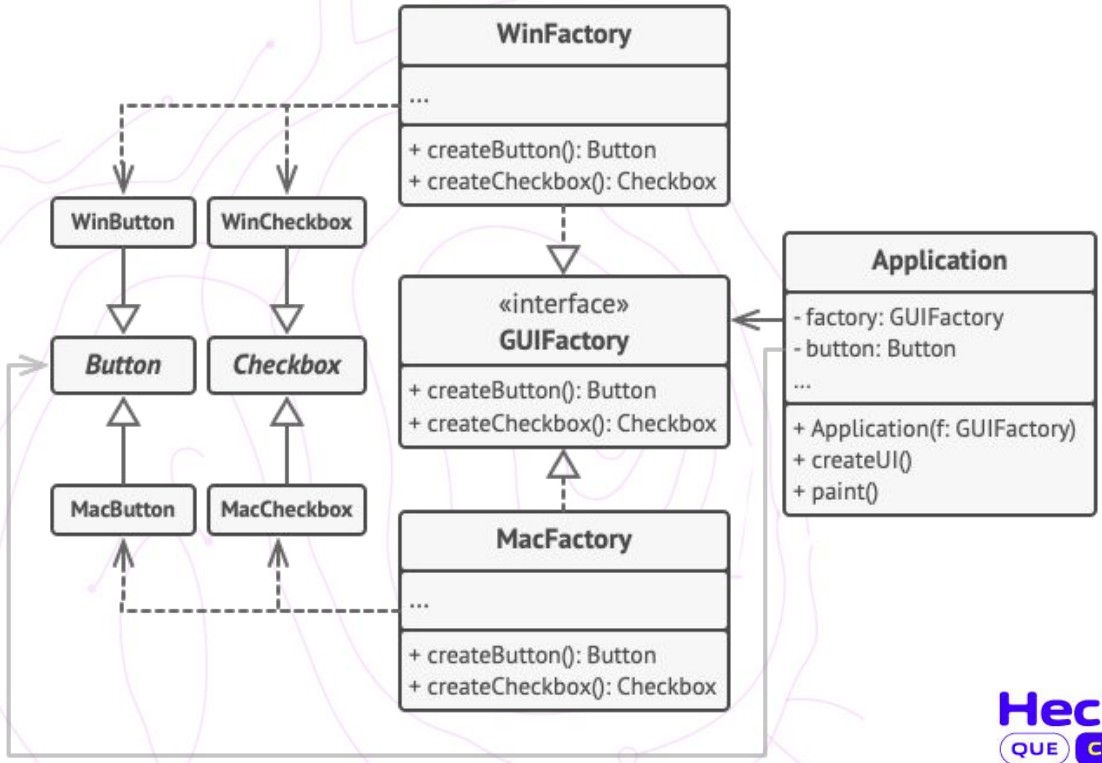
Desventajas

- Puede ser que el código se complique más de lo que debería, ya que se introducen muchas nuevas interfaces y clases junto al patrón.

Abstract Factory

Ejemplo

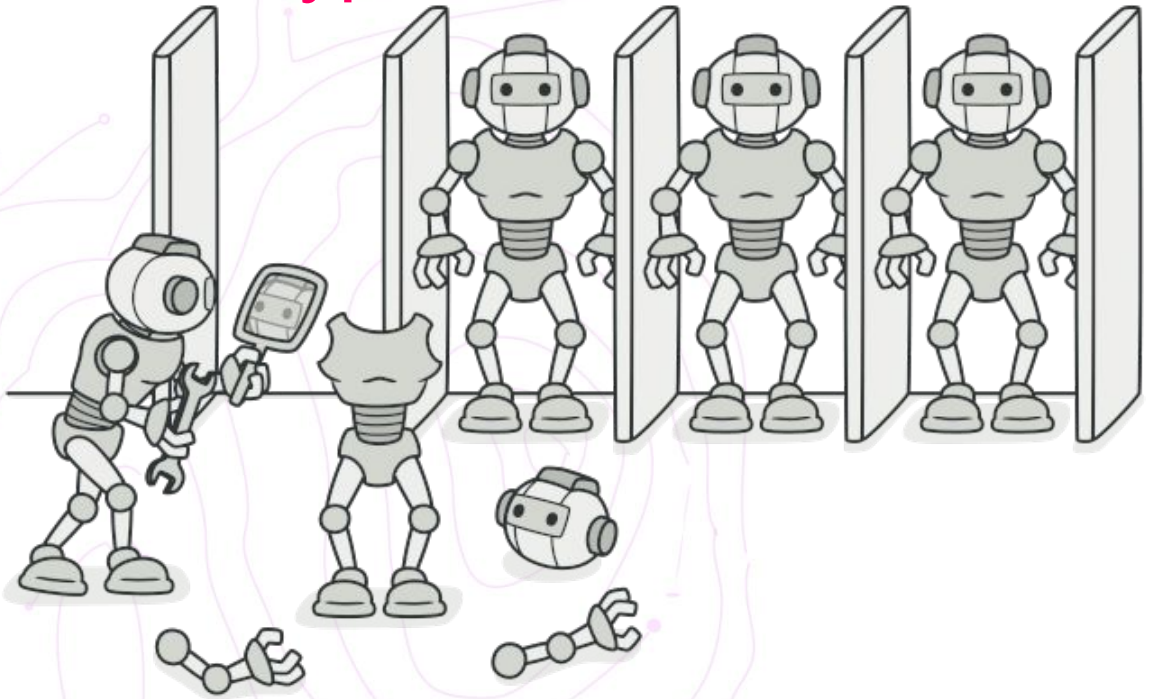




Prototype

Propósito

Es un patrón que nos permite copiar objetos existentes sin que el código dependa de sus clases.



Prototype

Problema

- Digamos que tienes un objeto y quieres crear una copia exacta de él. ¿Cómo lo harías?
- En primer lugar, debes crear un nuevo objeto de la misma clase. Después debes recorrer todos los campos del objeto original y copiar sus valores en el nuevo objeto.
- No todos los objetos se pueden copiar de este modo, porque algunos de los campos del objeto pueden ser privados e invisibles desde fuera del propio objeto.
- Dado que debes conocer la clase del objeto para crear un duplicado, el código se vuelve dependiente de esa clase.

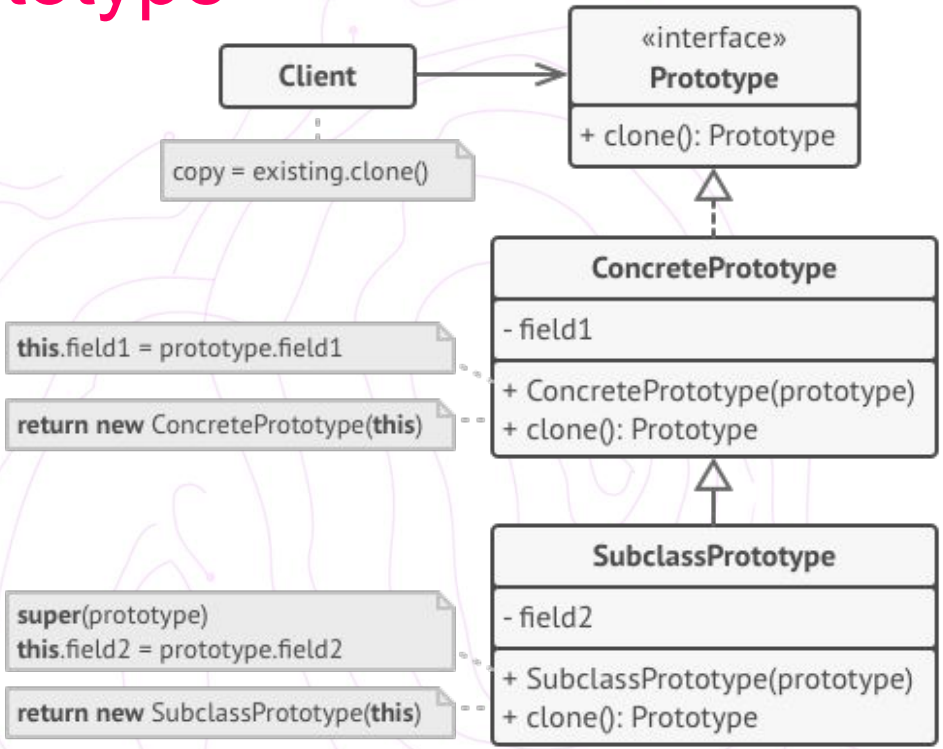
Prototype

Solución

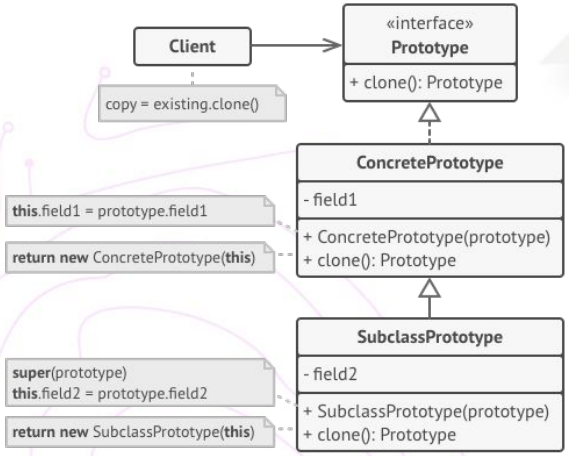
- El patrón Prototype delega el proceso de clonación a los propios objetos que están siendo clonados.
- El patrón declara una interfaz común para todos los objetos que soportan la clonación.
- Esta interfaz nos permite clonar un objeto sin acoplar el código a la clase de ese objeto. Normalmente, dicha interfaz contiene un único método clonar.
- Funciona así: se crea un grupo de objetos configurados de maneras diferentes. Cuando necesites un objeto como el que has configurado, clonas un prototipo en lugar de construir un nuevo objeto desde cero.

Prototype

Solución General



Prototype



Ventajas

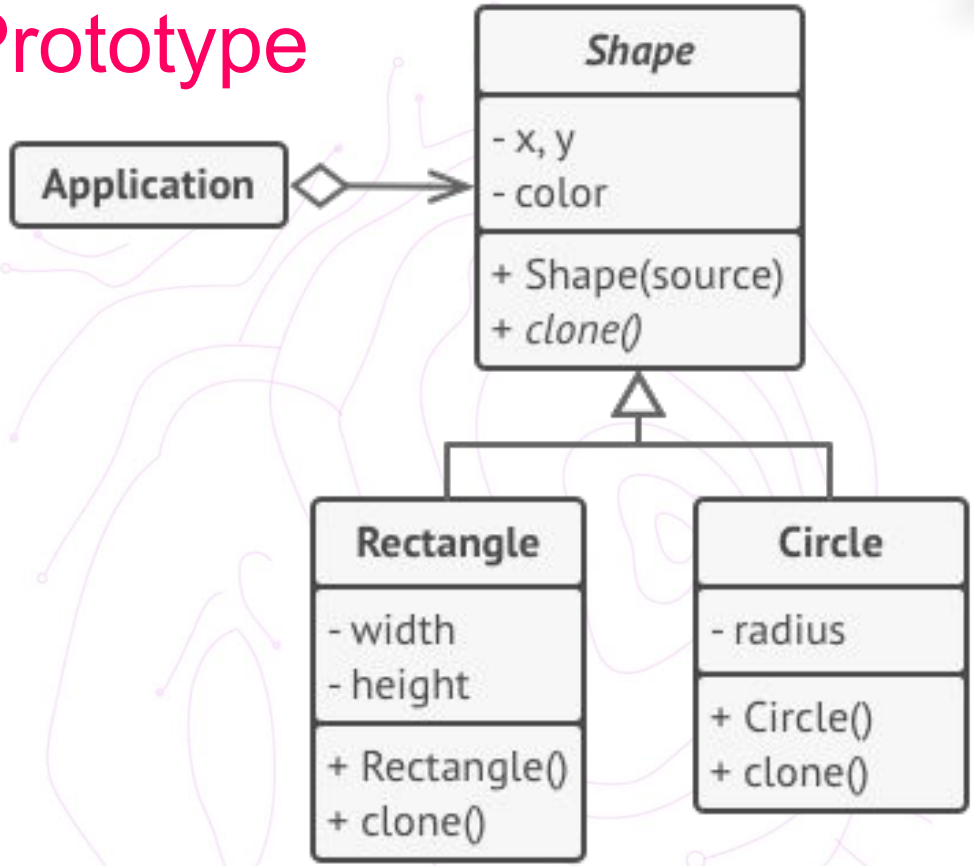
- Puedes clonar objetos sin acoplarlos a sus clases concretas.
- Puedes evitar un código de inicialización repetido clonando prototipos prefabricados.
- Puedes crear objetos complejos con más facilidad.
- Obtienes una alternativa a la herencia al tratar con preajustes de configuración para objetos complejos.

Desventajas

- Clonar objetos complejos con referencias circulares puede resultar complicado.

Prototype

Ejemplo



Universidad Tecnológica de Pereira

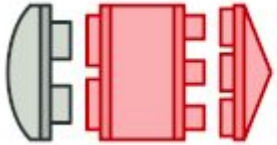
Mision TIC 2022

El futuro digital es de todos

MinTIC

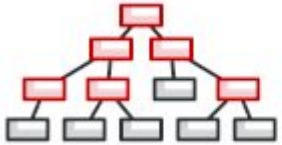
Patrones estructurales

Hechos QUE CONECTAN

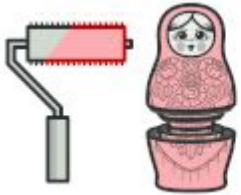


Adapter

Bridge

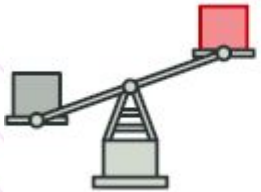


Composite

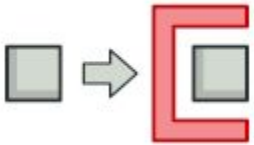


Decorator

Facade



Flyweight



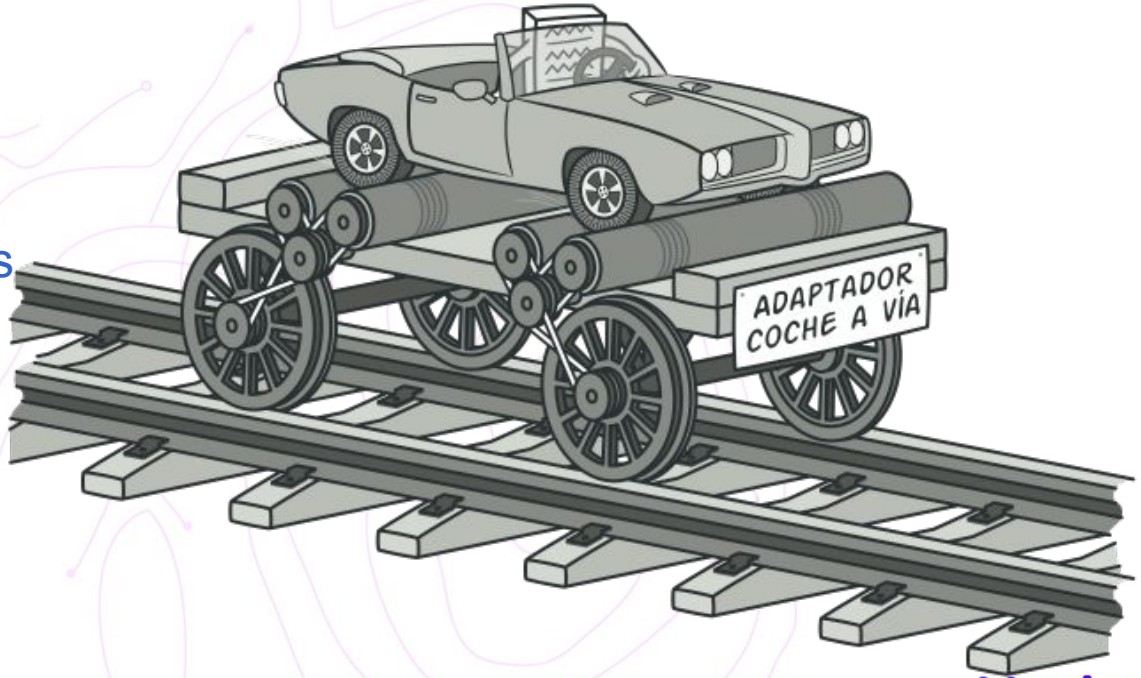
Proxy

Patrones estructurales

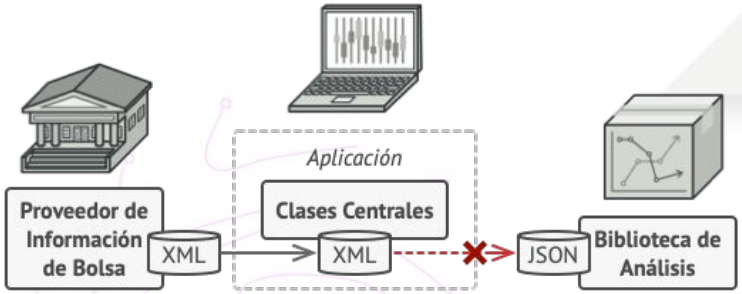
Adapter

Propósito

Es un patrón que permite la colaboración entre objetos con interfaces incompatibles



Adapter

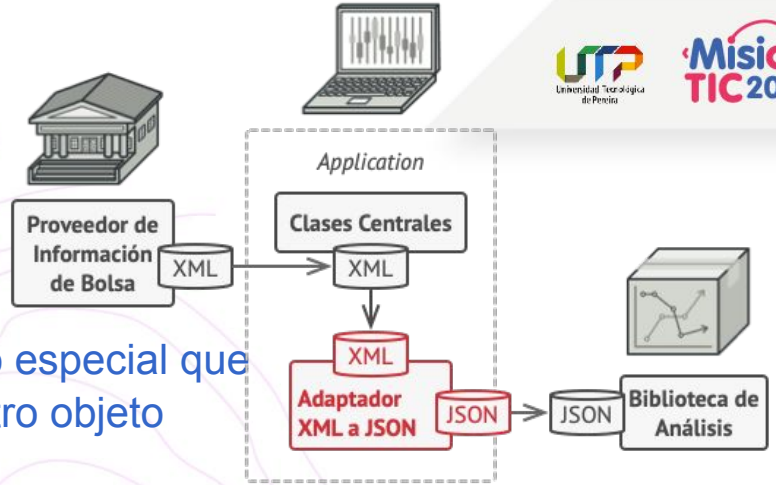


Problema

- Imagina que estás creando una aplicación de monitoreo del mercado de valores. La aplicación descarga la información de bolsa desde varias fuentes en formato XML para presentarla al usuario con bonitos gráficos y diagramas.
- En cierto momento, decides mejorar la aplicación integrando una inteligente biblioteca de análisis de una tercera persona. Pero hay una trampa: la biblioteca de análisis solo funciona con datos en formato JSON.
- Podrías cambiar la biblioteca para que funcione con XML. Sin embargo, esto podría descomponer parte del código existente que depende de la biblioteca. Y, lo que es peor, podrías no tener siquiera acceso al código fuente de la biblioteca, lo que hace imposible esta solución.

Adapter

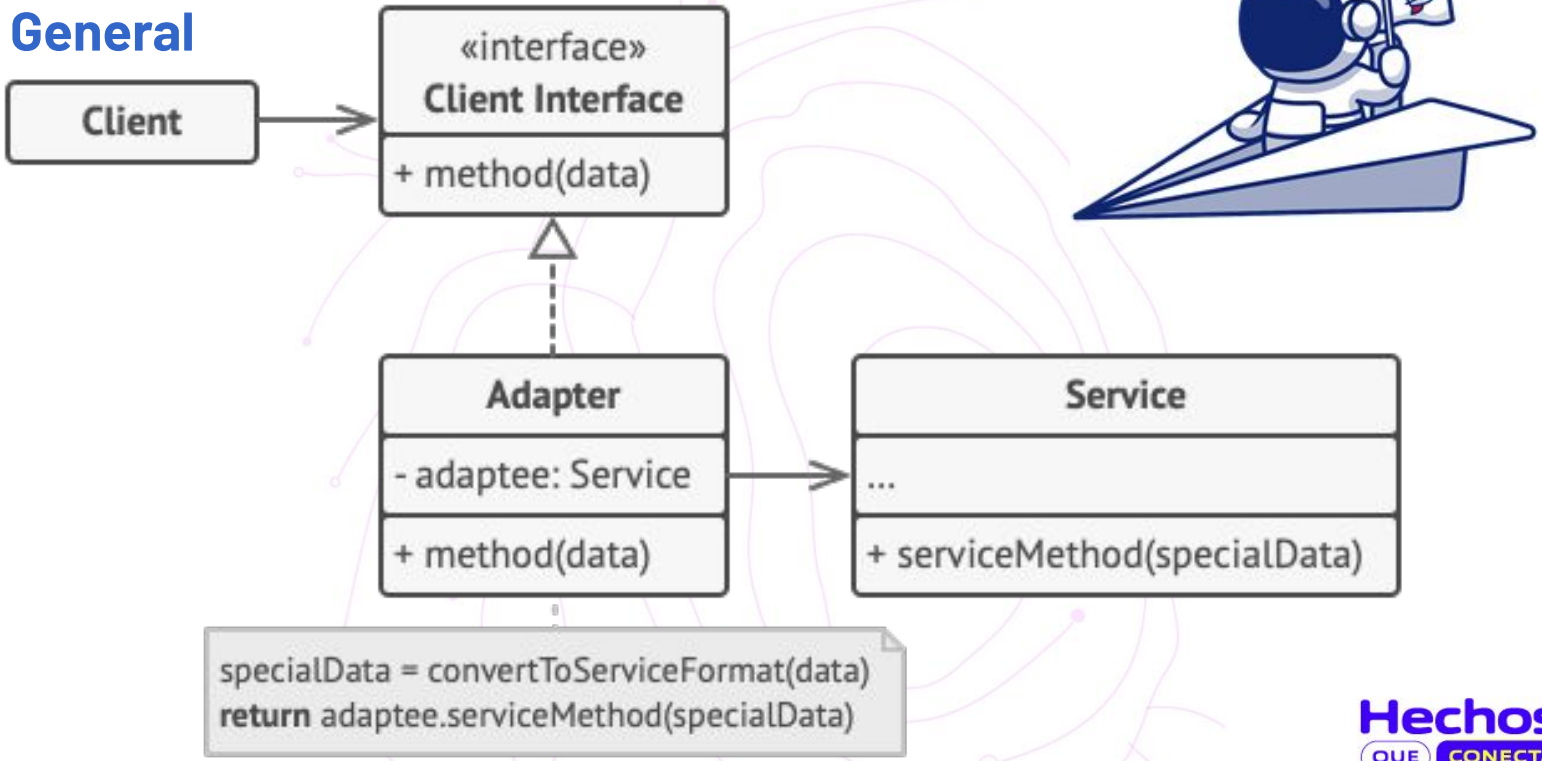
Solución



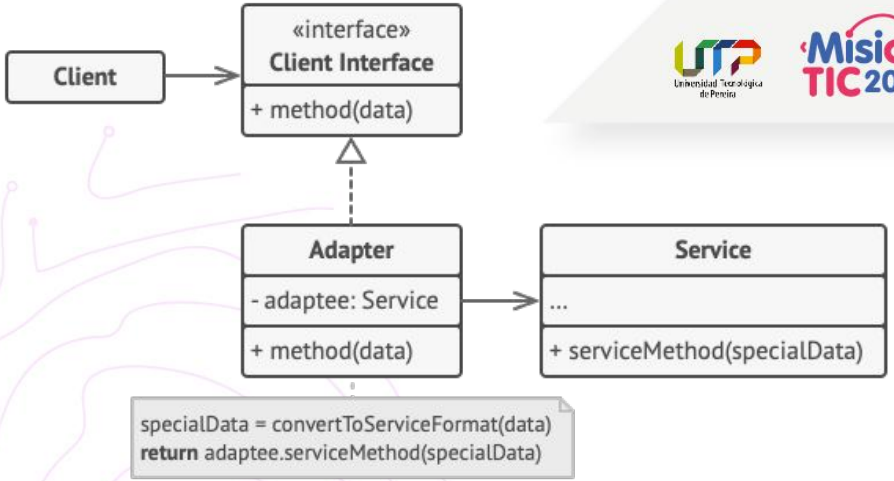
- Puedes crear un **adaptador**. Se trata de un objeto especial que convierte la interfaz de un objeto, de forma que otro objeto pueda comprenderla.
- Por ejemplo, puedes envolver un objeto que opera con metros y kilómetros con un adaptador que convierte todos los datos al sistema anglosajón, es decir, pies y millas.
- Funciona así:
  - El adaptador obtiene una interfaz compatible con uno de los objetos existentes.
  - Utilizando esta interfaz, el objeto existente puede invocar con seguridad los métodos del adaptador.
  - Al recibir una llamada, el adaptador pasa la solicitud al segundo objeto, pero en un formato y orden que ese segundo objeto espera.

Adapter

Solución General



Adapter



Ventajas

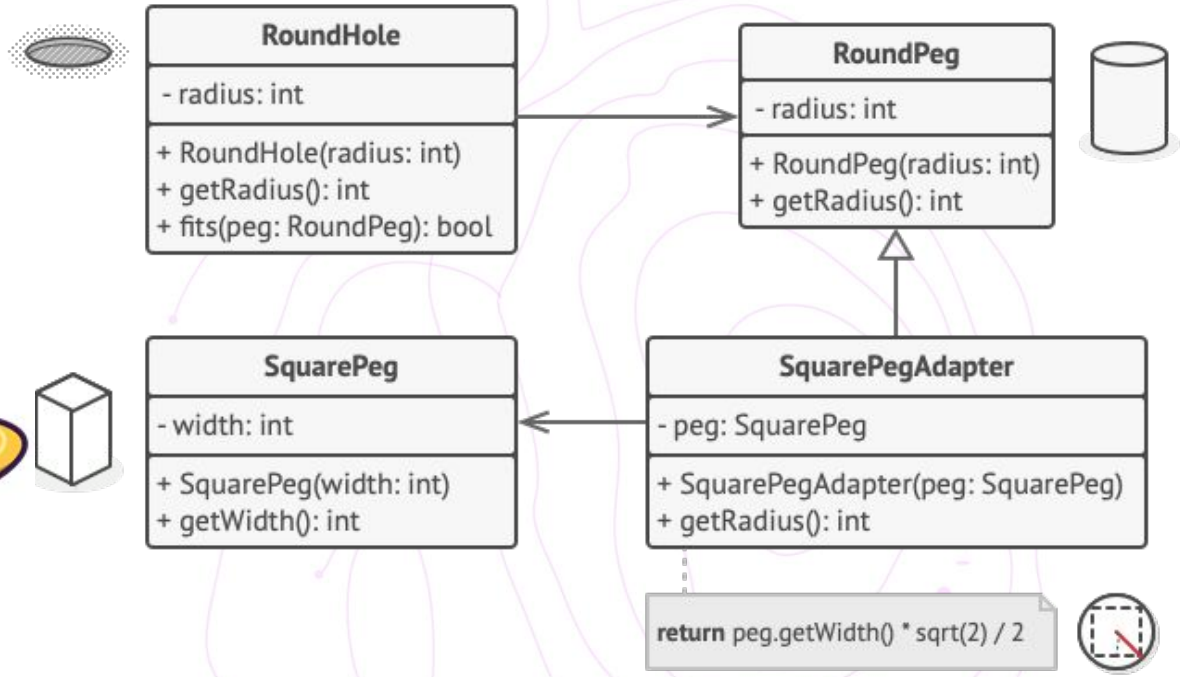
- ***Principio de responsabilidad única***. Puedes separar la interfaz o el código de conversión de datos de la lógica de negocio primaria del programa.
- ***Principio de abierto/cerrado.*** Puedes introducir nuevos tipos de adaptadores al programa sin descomponer el código cliente existente, siempre y cuando trabajen con los adaptadores a través de la interfaz con el cliente.
-

Desventajas

- La complejidad general del código aumenta, ya que debes introducir un grupo de nuevas interfaces y clases. En ocasiones resulta más sencillo cambiar la clase de servicio de modo que coincida con el resto de tu código.

Adapter

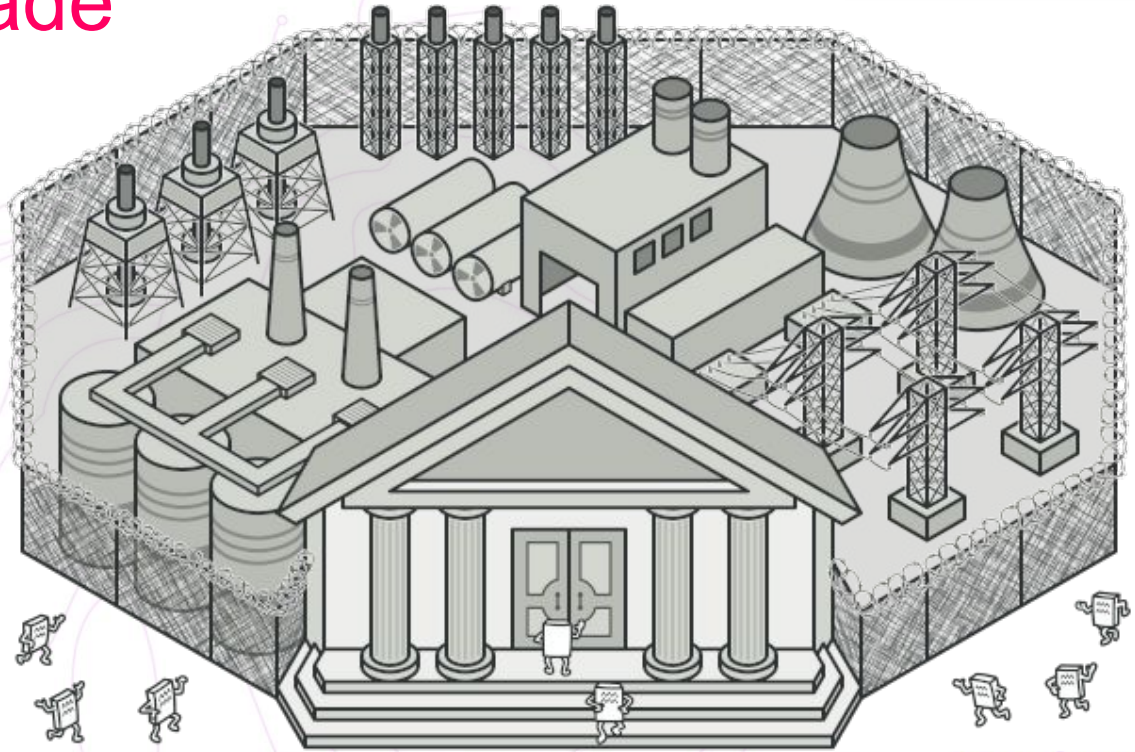
Ejemplo



Facade

Propósito

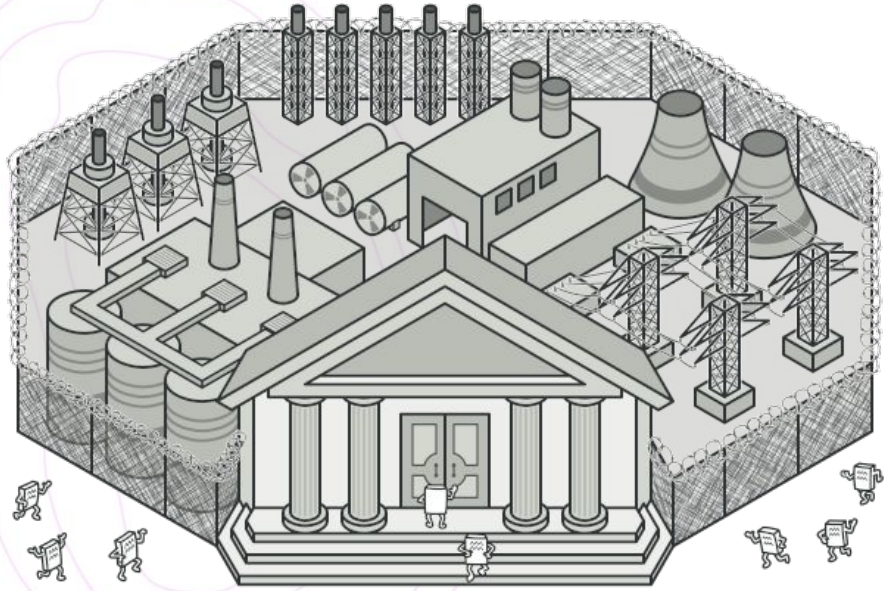
Es un patrón que proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases.



Facade

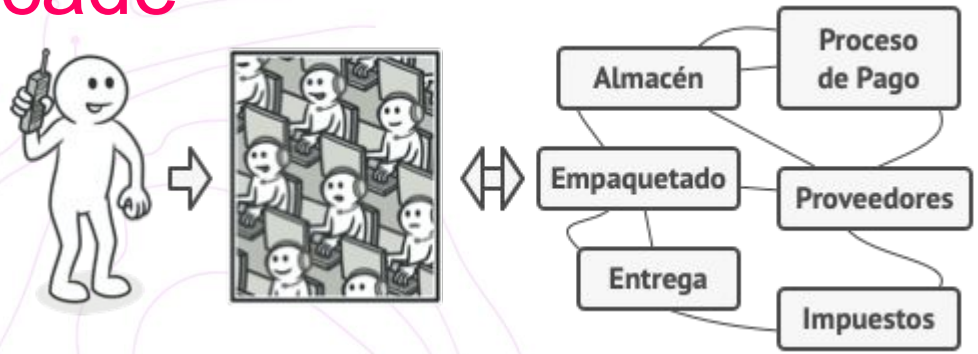
Problema

- Imagina que debes lograr que tu código trabaje con una sofisticada biblioteca o framework. Normalmente, debes inicializar todos esos objetos, llevar un registro de las dependencias, ejecutar los métodos en el orden correcto y así sucesivamente.
- Como resultado, la lógica de negocio de tus clases se vería estrechamente acoplada a los detalles de implementación de las clases de terceros, haciéndola difícil de comprender y mantener.



Facade

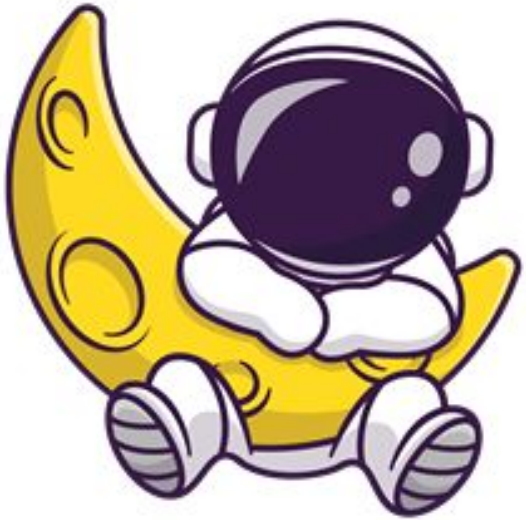
Solución

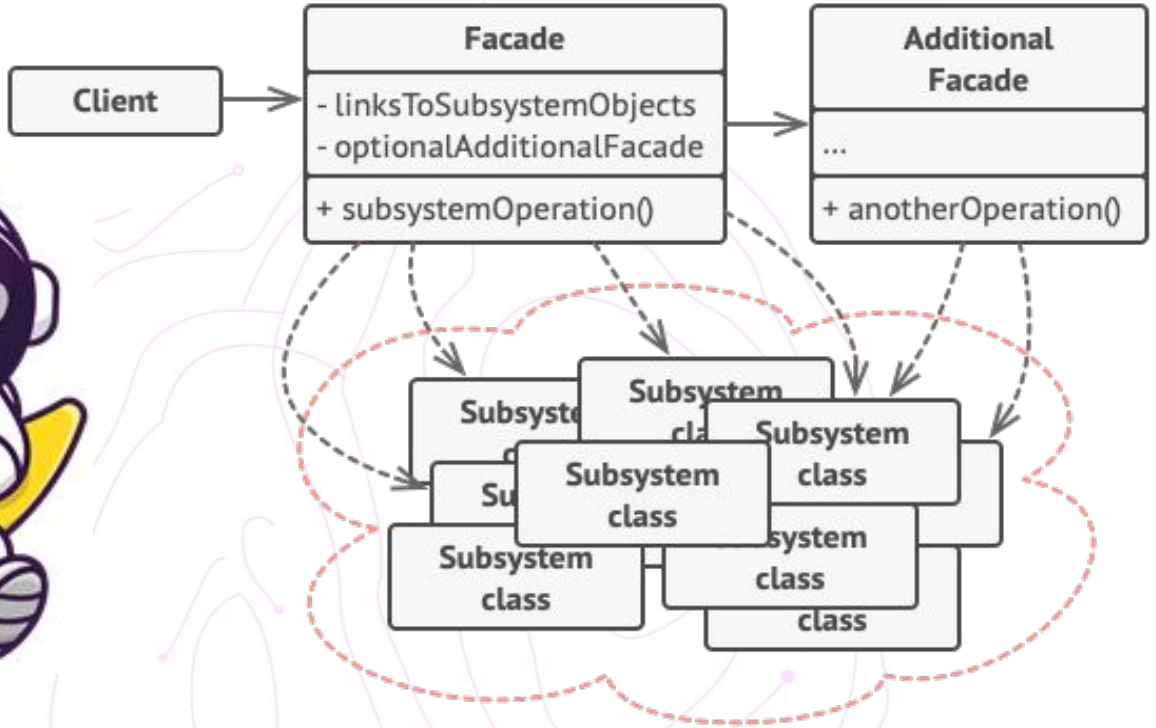


- Una fachada es una clase que proporciona una interfaz simple a un subsistema complejo que contiene muchas partes móviles.
- Una fachada puede proporcionar una funcionalidad limitada en comparación con trabajar directamente con el subsistema. Sin embargo, tan solo incluye las funciones realmente importantes para los clientes.
- Tener una fachada resulta útil cuando tienes que integrar tu aplicación con una biblioteca sofisticada con decenas de funciones, de la cual sólo necesitas una pequeña parte.

Facade

Solución General





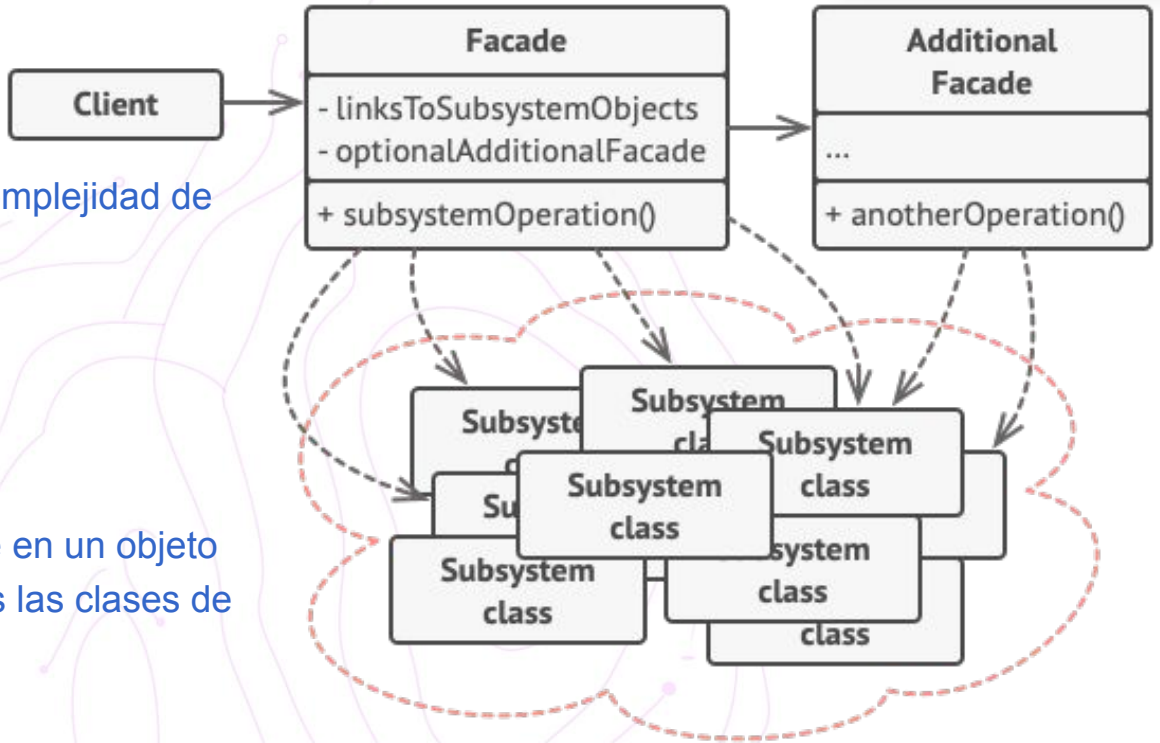
Facade

Ventajas

- Puedes aislar tu código de la complejidad de un subsistema.

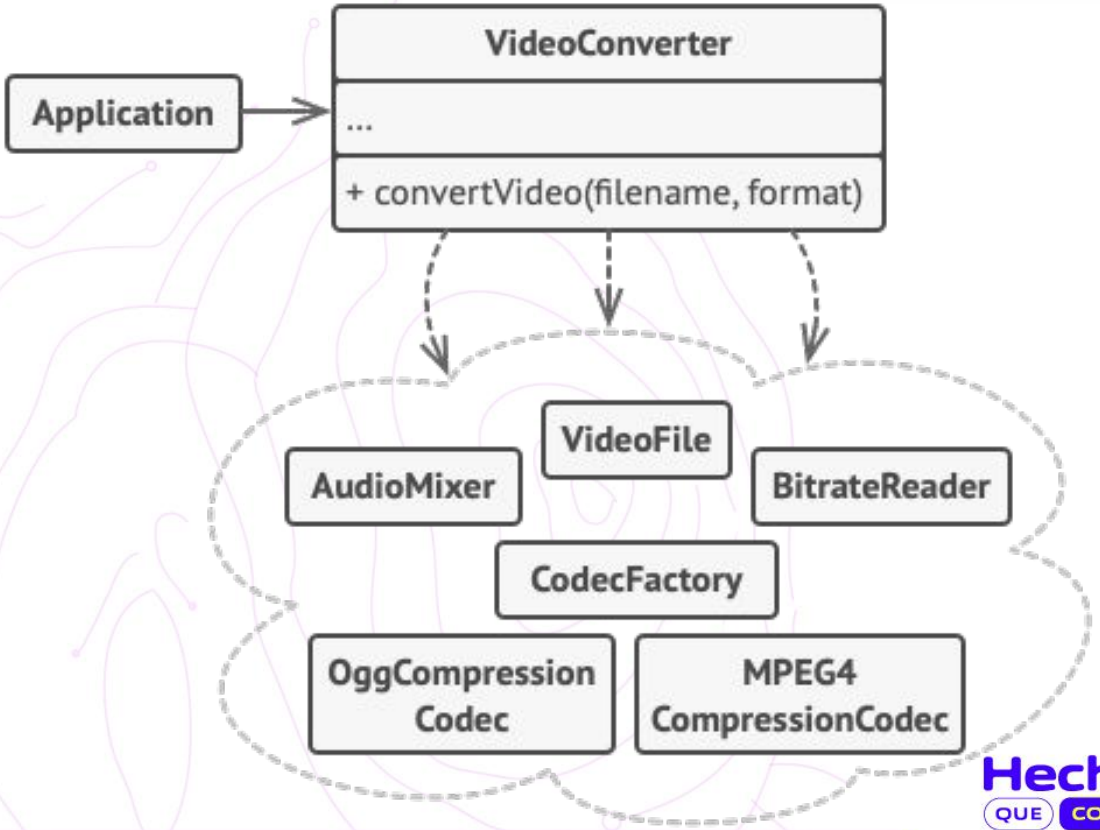
Desventajas

- Una fachada puede convertirse en un objeto todopoderoso acoplado a todas las clases de una aplicación.



Facade

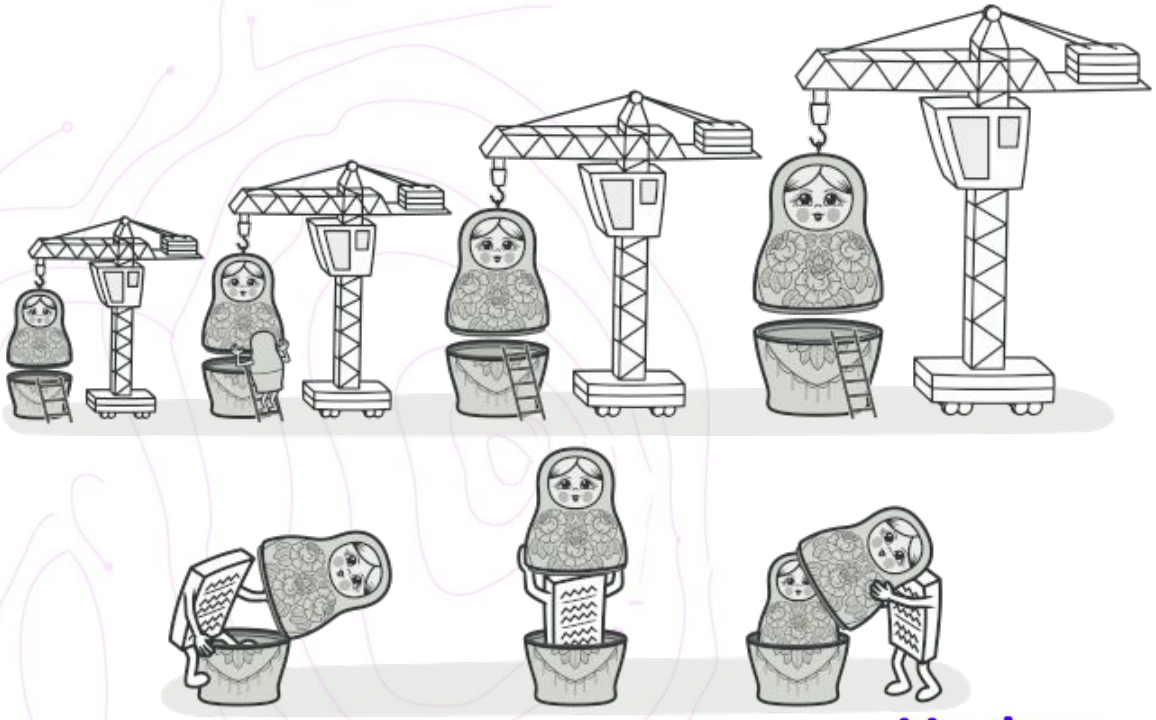
Ejemplo



Decorator

Propósito

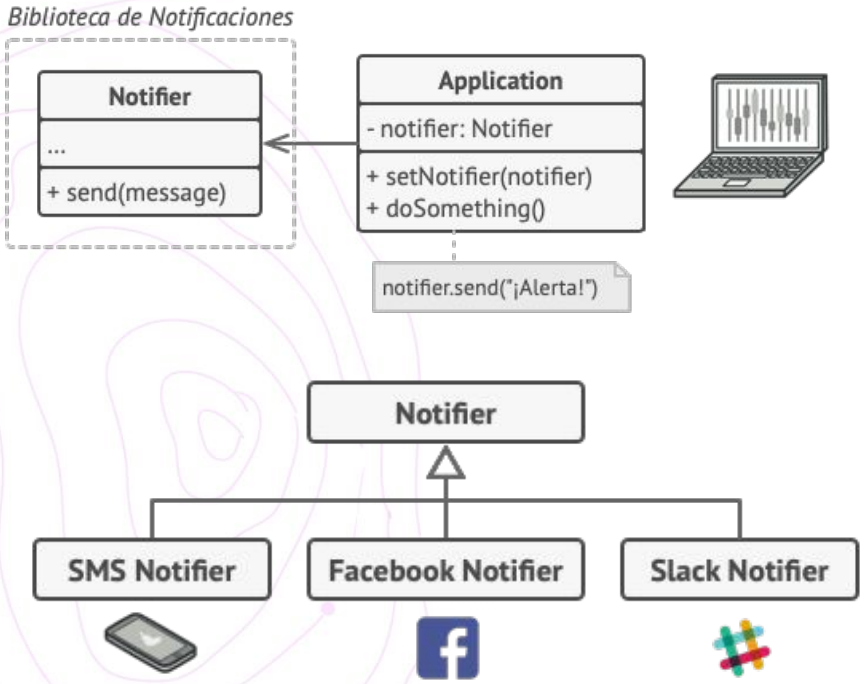
Es un patrón que te permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.



Decorator

Problema

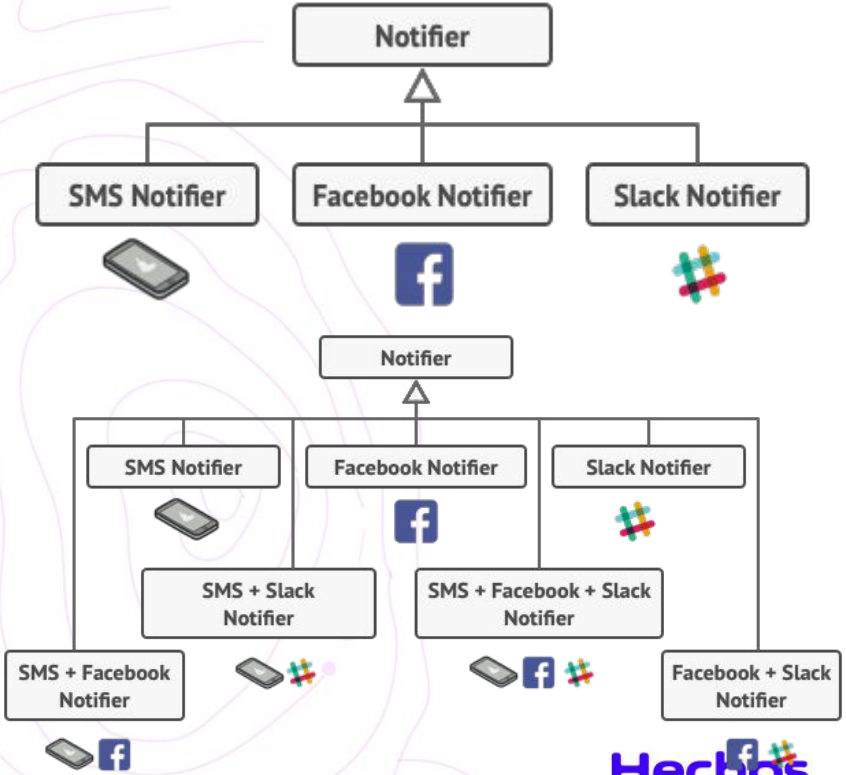
- Imagina que estás trabajando en una **biblioteca** de notificaciones que permite a otros programas notificar a sus usuarios acerca de eventos importantes.
- Te das cuenta de que los usuarios de la biblioteca esperan algo más que unas simples notificaciones por correo. A muchos de ellos les gustaría recibir mensajes ***SMS*** sobre asuntos importantes. Otros querrían recibir las notificaciones por ***Facebook*** y, por supuesto, a los usuarios corporativos les encantaría recibir notificaciones por ***Slack***.



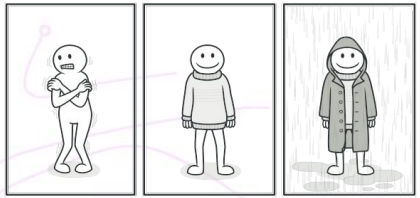
Decorator

Problema

- Pero alguien te hace una pregunta razonable: "***¿Por qué no se pueden utilizar varios tipos de notificación al mismo tiempo?*** Si tu casa está en llamas, probablemente quieras que te informen a través de todos los canales".
- Intentaste solucionar ese problema creando subclases especiales que combinan varios métodos de notificación dentro de una clase.
- Sin embargo, enseguida resultó evidente que esta solución incrementaría el código en gran medida, no sólo el de la biblioteca, sino también el código cliente.

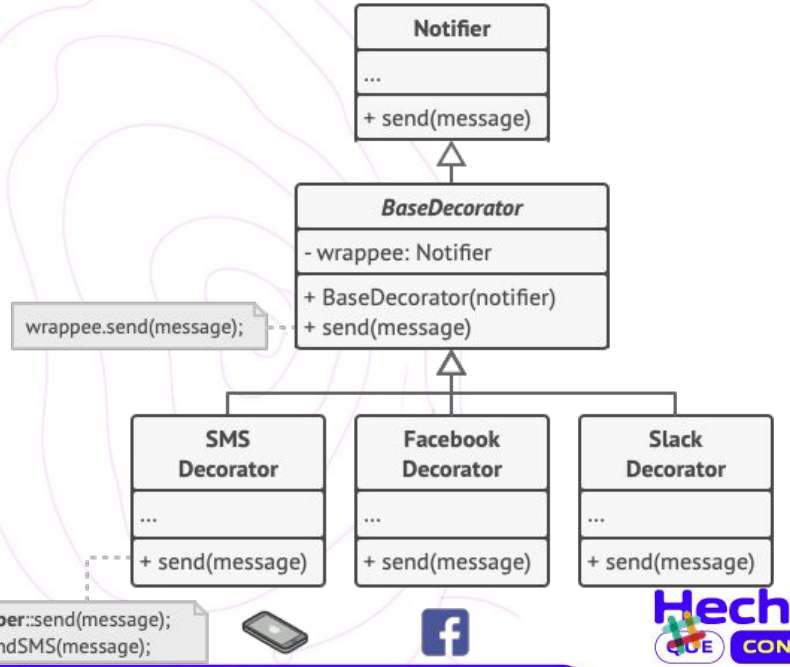


Decorator

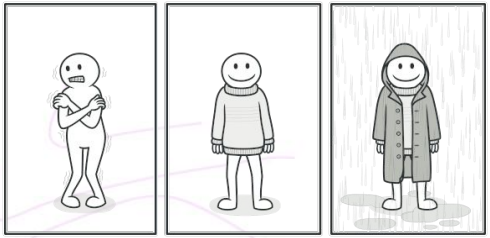


Solución

- Cuando tenemos que alterar la funcionalidad de un objeto, lo primero que se viene a la mente es extender una clase.
  - La herencia es estática.
  - Las subclases sólo pueden tener una clase padre.
- "***Wrapper***" (envoltorio, en inglés) es el sobrenombre alternativo del patrón Decorator.

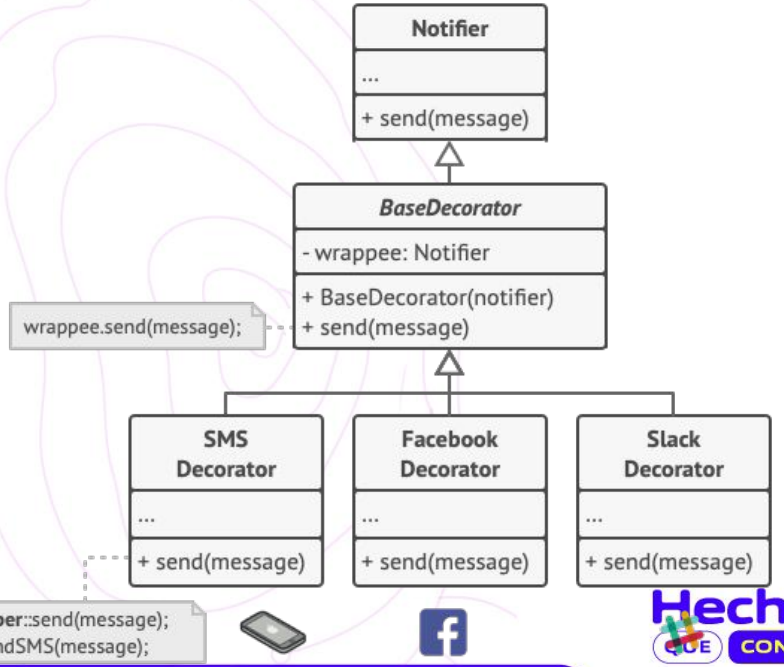


Decorator



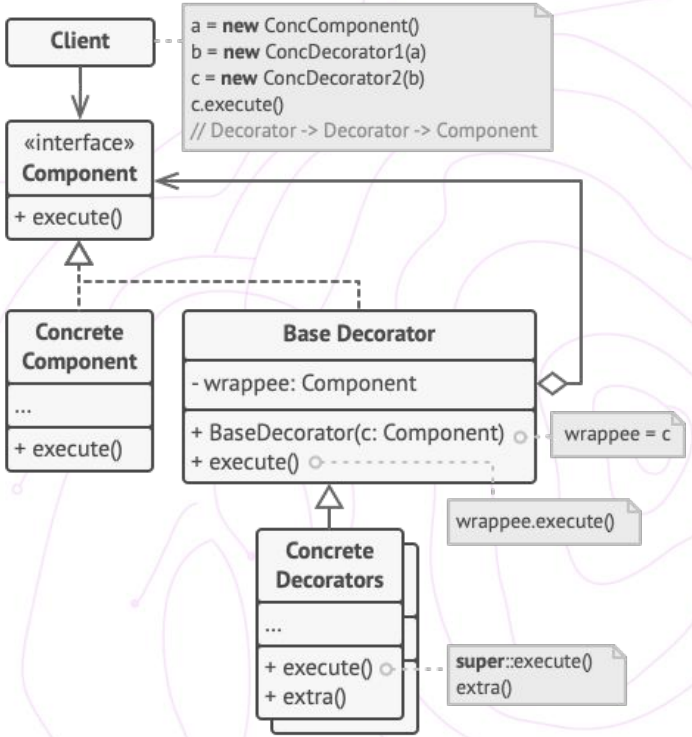
Solución

- Un ***wrapper*** es un objeto que puede vincularse con un objeto objetivo.
- El wrapper contiene el mismo grupo de métodos que el objetivo y le delega todas las solicitudes que recibe.
- No obstante, el wrapper puede alterar el resultado haciendo algo antes o después de pasar la solicitud al objetivo.



Decorator

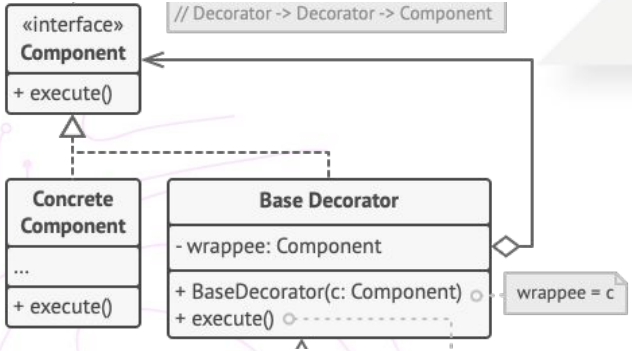
Solución General



Decorator

Ventajas

- Puedes extender el comportamiento de un objeto sin crear una nueva subclase.
- Puedes añadir o eliminar responsabilidades de un objeto durante el tiempo de ejecución.
- Puedes combinar varios comportamientos envolviendo un objeto con varios decoradores.
- ***Principio de responsabilidad única***. Puedes dividir una clase monolítica que implementa muchas variantes posibles de comportamiento, en varias clases más pequeñas.

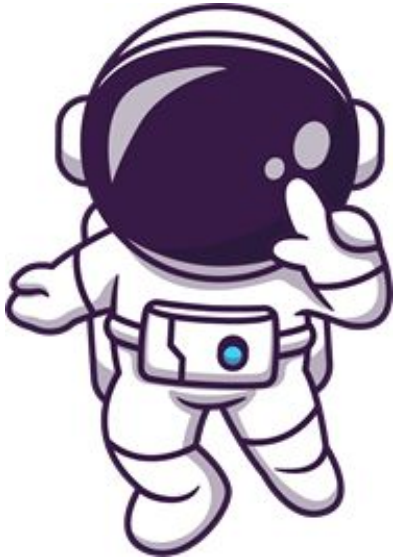


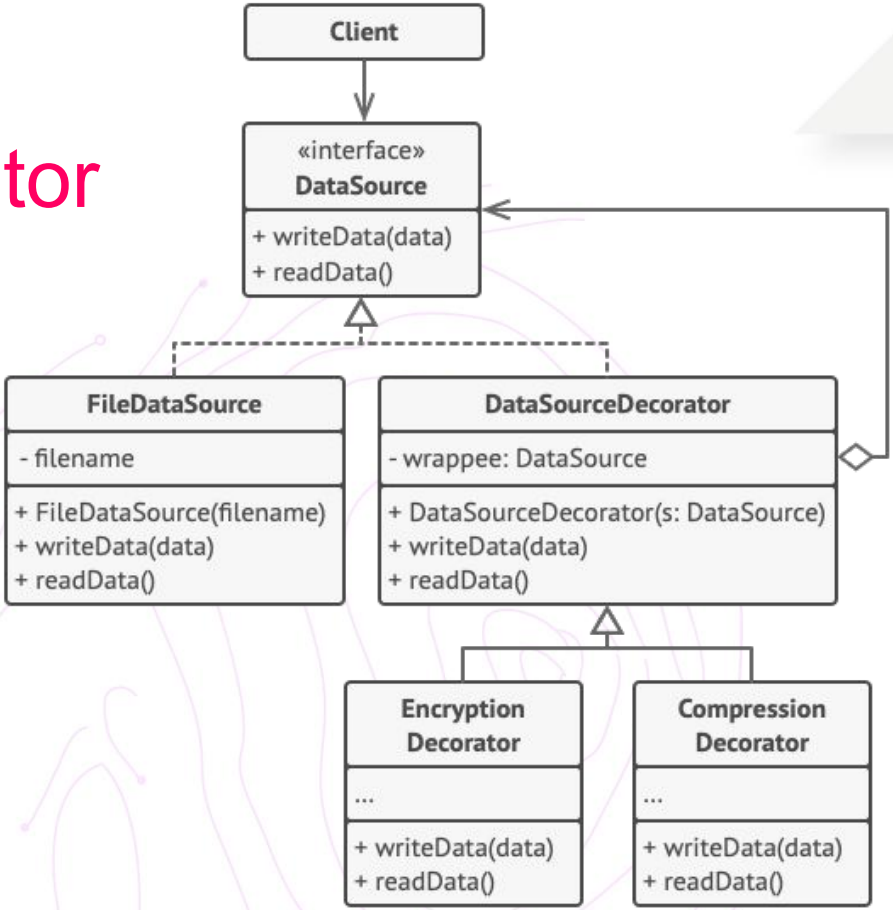
Desventajas

- Resulta difícil eliminar un wrapper específico de la pila de wrappers.
- Es difícil implementar un decorador de tal forma que su comportamiento no dependa del orden en la pila de decoradores.
- El código de configuración inicial de las capas pueden tener un aspecto desagradable.

Decorator

Ejemplo

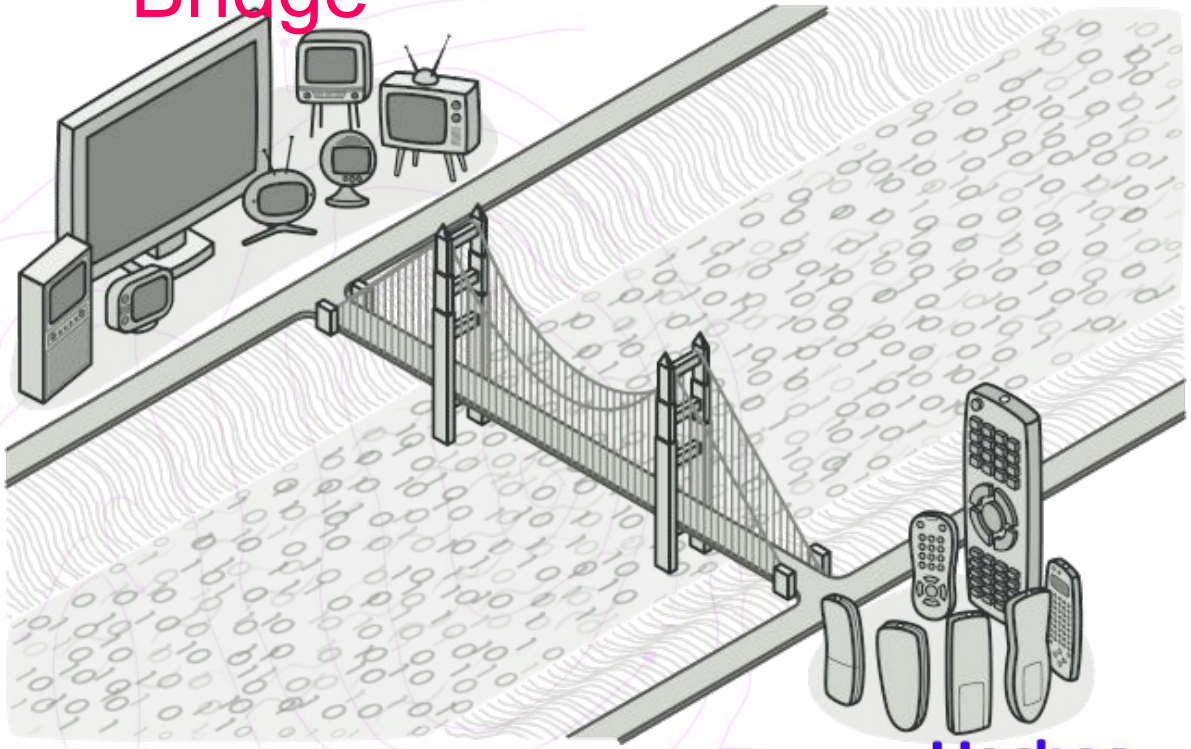




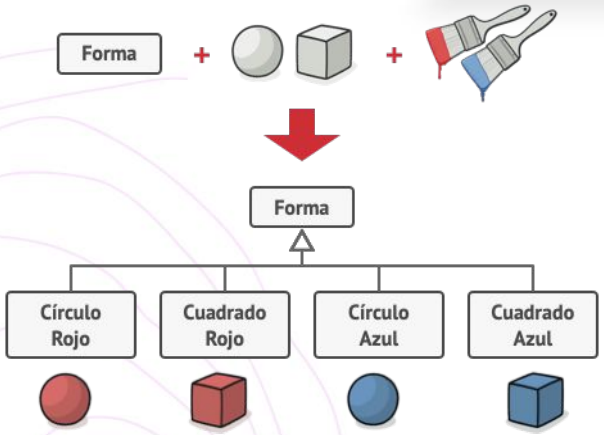
Bridge

Propósito

Es un patrón que te permite dividir una clase grande, o un grupo de clases estrechamente relacionadas, en dos jerarquías separadas (abstracción e implementación) que pueden desarrollarse de forma independiente la una de la otra.



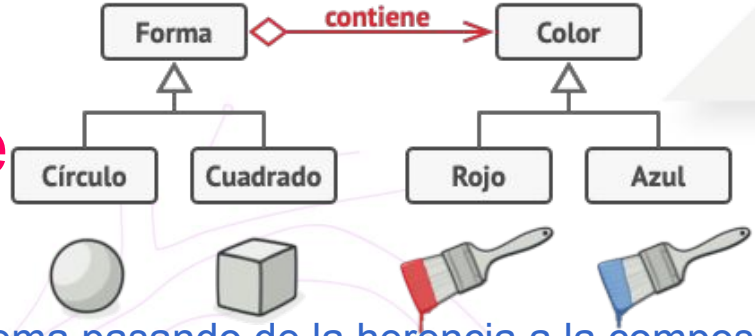
Bridge



Problema

- Digamos que tienes una clase geométrica Forma con un par de subclases: Círculo y Cuadrado.
- Deseas extender esta jerarquía de clase para que incorpore colores, por lo que planeas crear las subclases de forma Rojo y Azul.
- Añadir nuevos tipos de forma y color a la jerarquía hará que ésta crezca exponencialmente.
- Por ejemplo, para añadir una forma de triángulo deberás introducir dos subclases, una para cada color. Y, después, para añadir un nuevo color habrá que crear tres subclases, una para cada tipo de forma. Cuanto más avancemos, peor será.

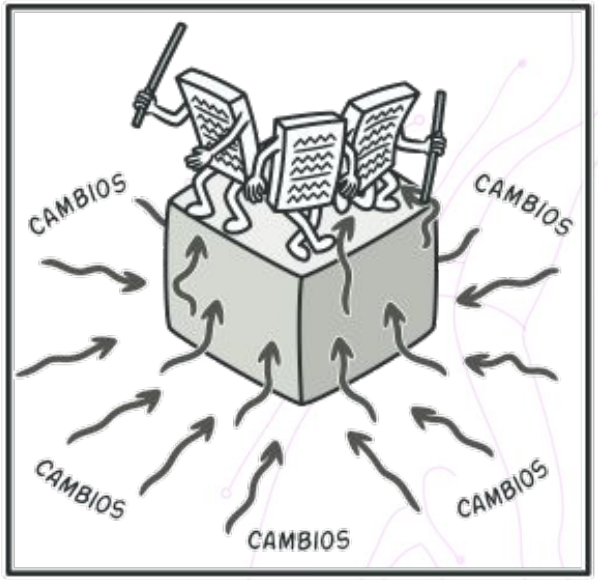
Bridge

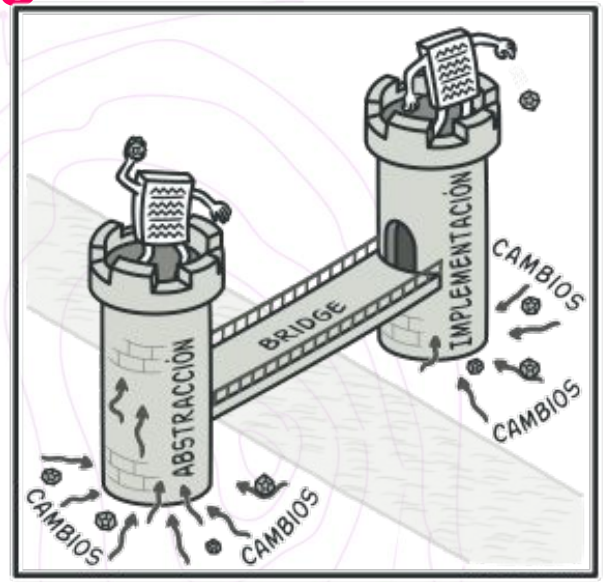


Solución

- El patrón intenta resolver este problema pasando de la herencia a la composición del objeto. Esto quiere decir que se extrae una de las dimensiones a una jerarquía de clases separada, de modo que las clases originales referencian un objeto de la nueva jerarquía, en lugar de tener todo su estado y sus funcionalidades dentro de una clase.
- La clase Forma obtiene entonces un campo de referencia que apunta a uno de los objetos de color. Ahora la forma puede delegar cualquier trabajo relacionado con el color al objeto de color vinculado.
- Esa referencia actuará como un puente entre las clases Forma y Color. En adelante, añadir nuevos colores no exigirá cambiar la jerarquía de forma y viceversa.

Bridge

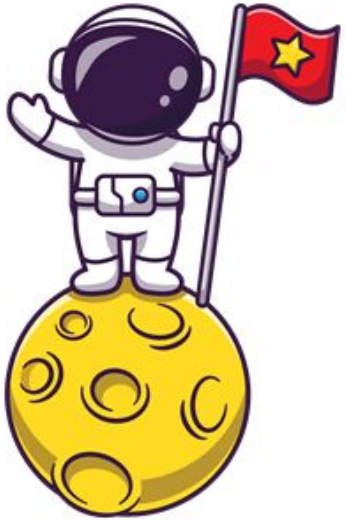


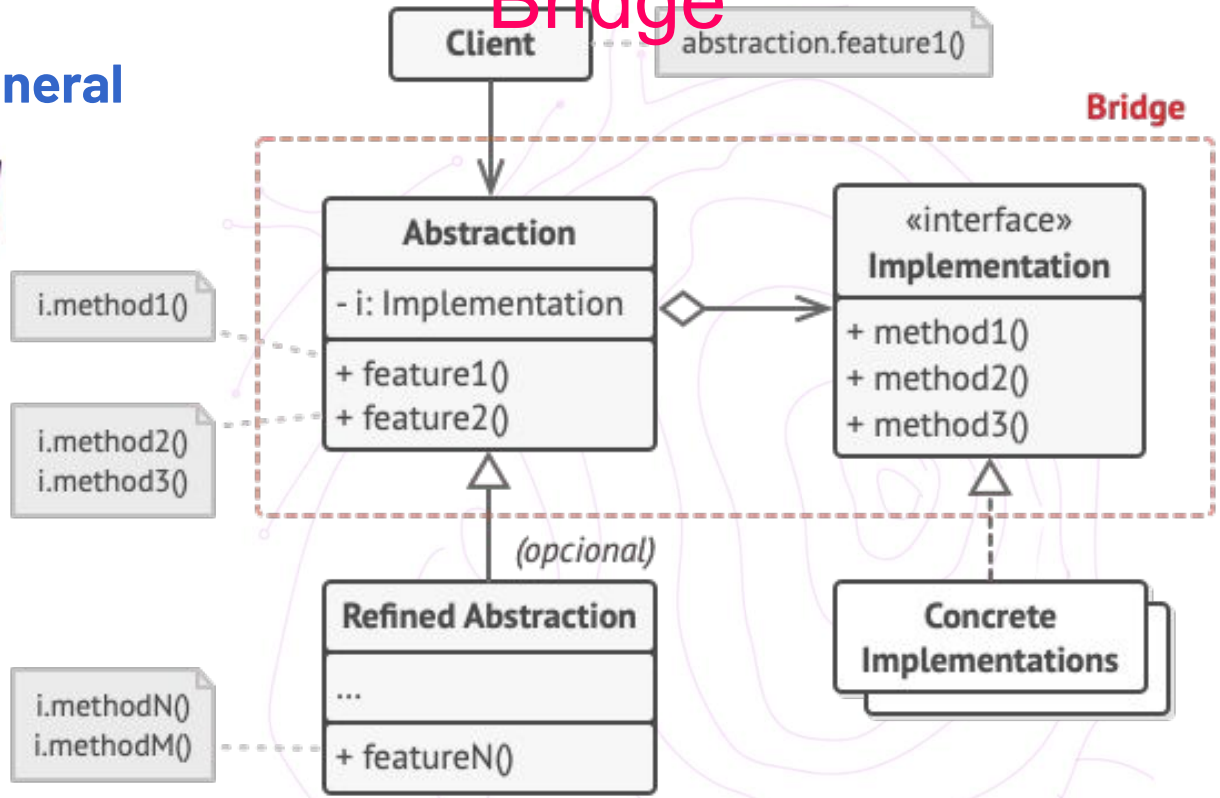


La **Abstracción** (también llamada interfaz) es una capa de control de alto nivel para una entidad. Esta capa no tiene que hacer ningún trabajo real por su cuenta, sino que debe delegar el trabajo a la capa de **implementación** (también llamada plataforma).

Bridge

Solución General

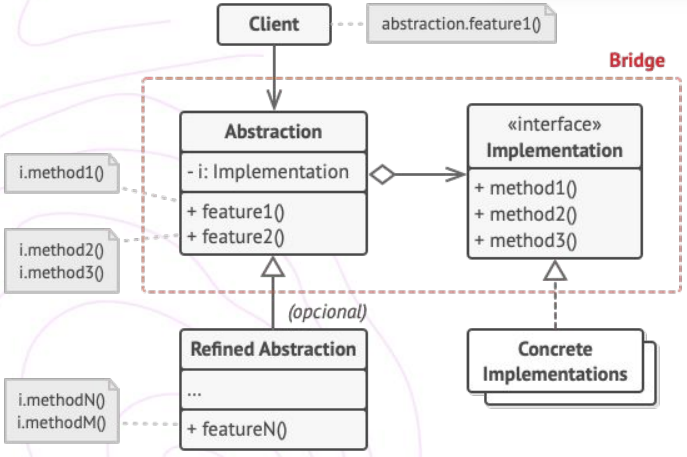




Bridge

Ventajas

- Puedes crear clases y aplicaciones independientes de plataforma.
- El código cliente funciona con abstracciones de alto nivel. No está expuesto a los detalles de la plataforma.
- ***Principio de abierto/cerrado***. Puedes introducir nuevas abstracciones e implementaciones independientes entre sí.
- ***Principio de responsabilidad única***. Puedes centrarte en la lógica de alto nivel en la abstracción y en detalles de la plataforma en la implementación.
-



Desventajas

- Puede ser que el código se complique si aplicas el patrón a una clase muy cohesionada.

Bridge

Ejemplo

```
if (device.isEnabled())
  device.disable()
else
  device.enable()
```

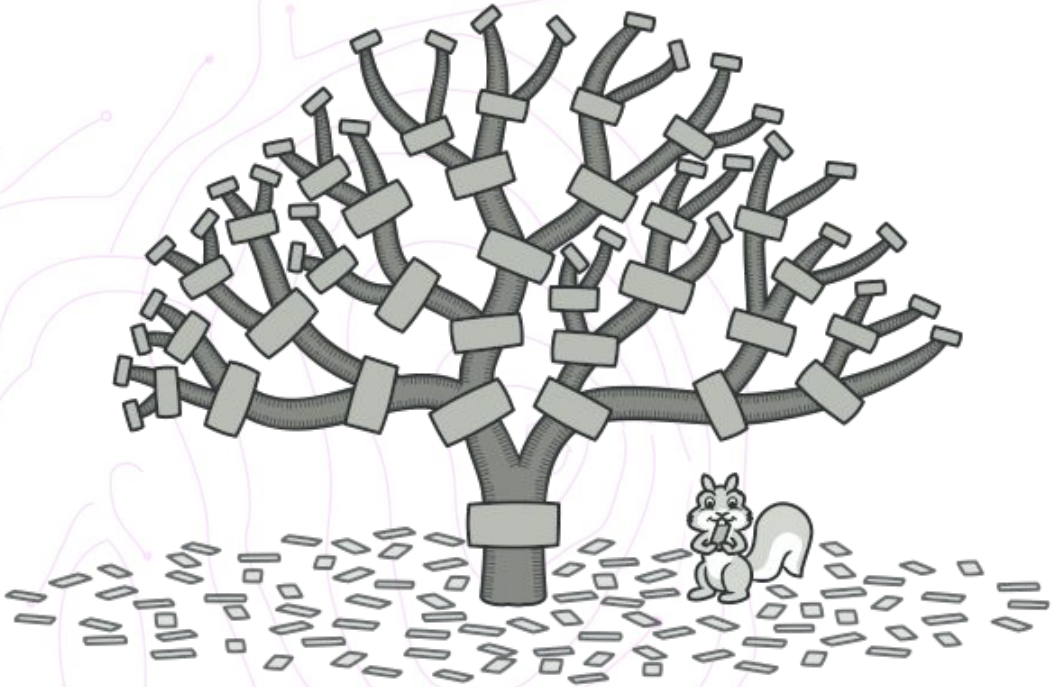
```
old = device.getChannel()
device.setChannel(old+1)
```

```
device.setVolume(0)
```


Composite

Propósito

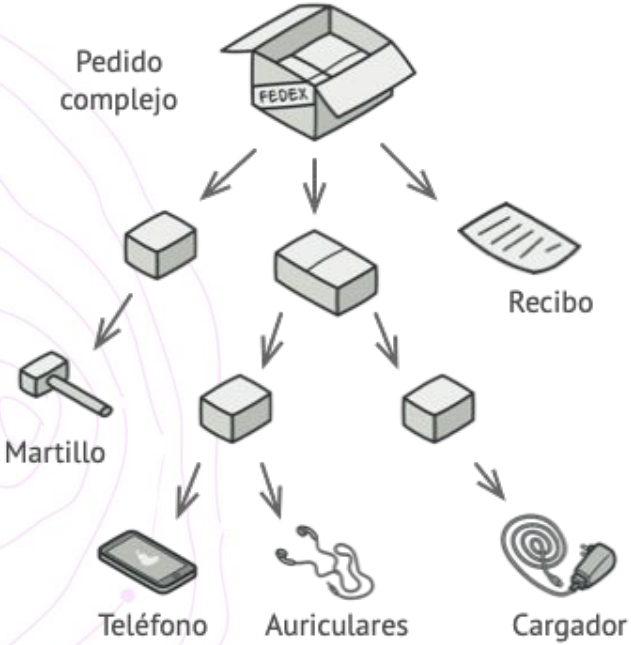
Es un patrón que te permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.



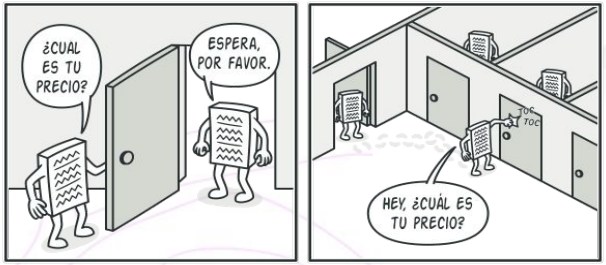
Composite

Problema

- Imagina que tienes dos tipos de objetos: Productos y Cajas. Una Caja puede contener varios Productos así como cierto número de Cajas más pequeñas. Estas Cajas pequeñas también pueden contener algunos Productos o incluso Cajas más pequeñas, y así sucesivamente.
- Digamos que decides crear un sistema de pedidos que utiliza estas clases. Los pedidos pueden contener productos sencillos sin envolver, así como cajas llenas de productos... y otras cajas. ¿***Cómo determinarás el precio total de ese pedido***?



Composite

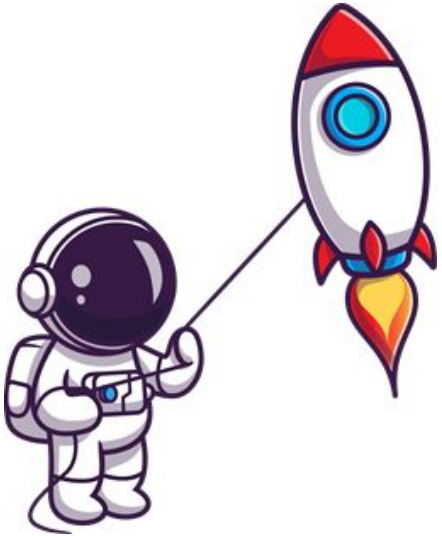


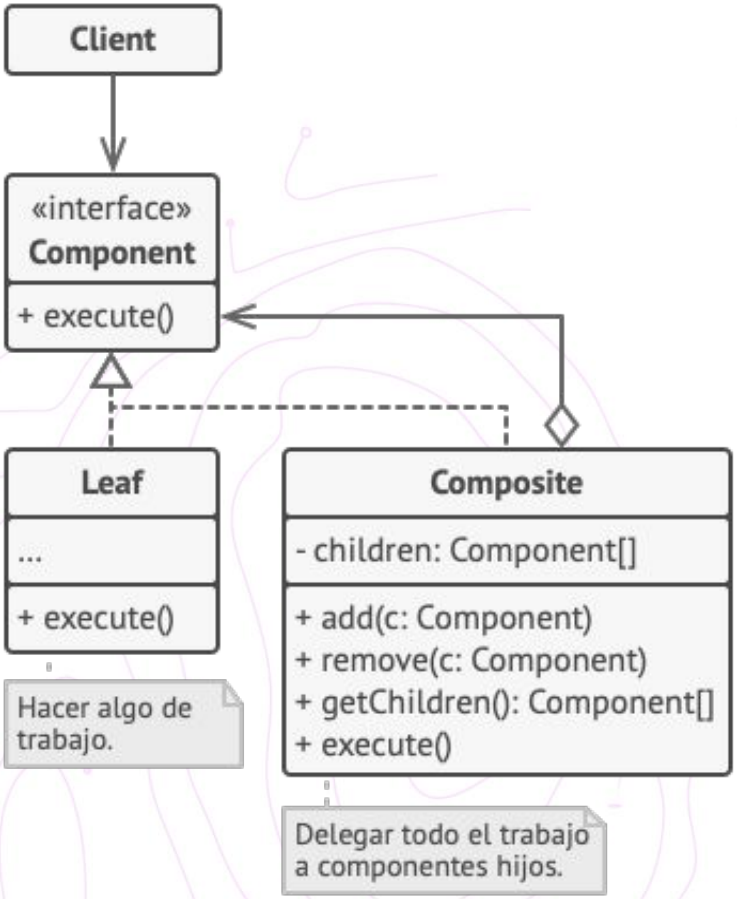
Solución

- El patrón sugiere que trabajes con Productos y Cajas a través de una interfaz común que declara un método para calcular el precio total.
- Para un producto, sencillamente devuelve el precio del producto.
- Para una caja, recorre cada artículo que contiene la caja, pregunta su precio y devuelve un total por la caja. Si uno de esos artículos fuera una caja más pequeña, esa caja también comenzaría a repasar su contenido y así sucesivamente, hasta que se calcule el precio de todos los componentes internos.
- Una caja podría incluso añadir costos adicionales al precio final, como costos de empaquetado.
- No tienes que saber si un objeto es un producto simple o una sofisticada caja. Puedes tratarlos a todos por igual a través de la interfaz común.

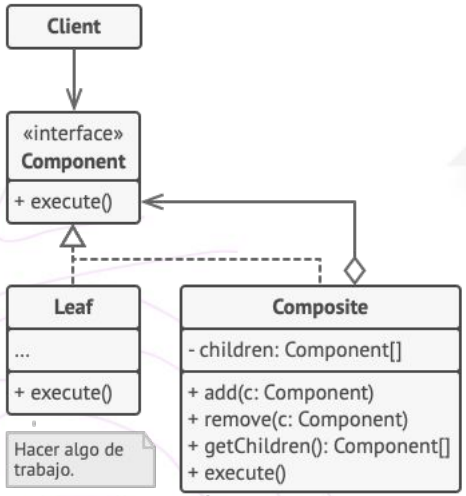
Composite

Solución General





Composite



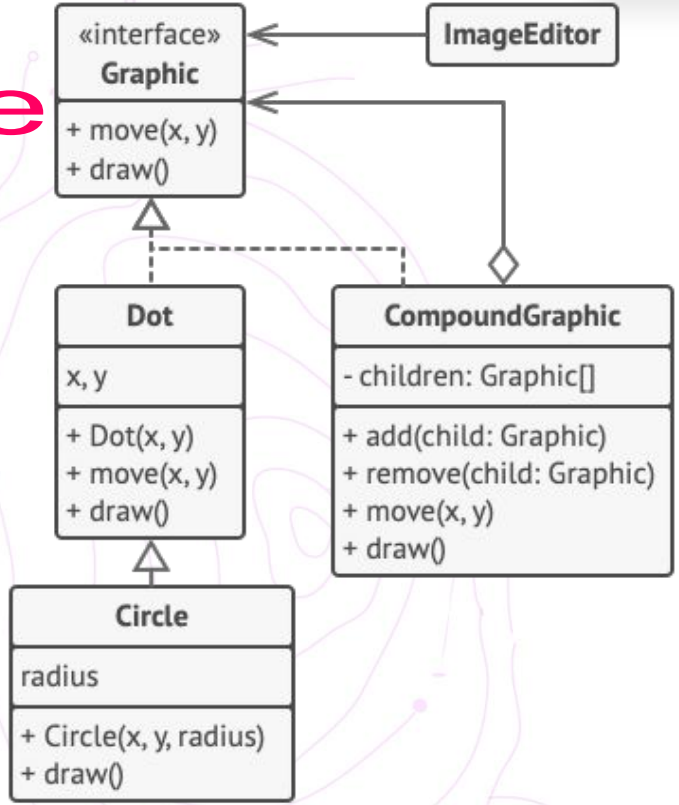
Ventajas

- Puedes trabajar con estructuras de árbol complejas con mayor comodidad: utiliza el polimorfismo y la recursión en tu favor.
- ***Principio de abierto/cerrado***. Puedes introducir nuevos tipos de elemento en la aplicación sin descomponer el código existente, que ahora funciona con el árbol de objetos.

Desventajas

- Puede resultar difícil proporcionar una interfaz común para clases cuya funcionalidad difiere demasiado. En algunos casos, tendrás que generalizar en exceso la interfaz componente, provocando que sea más difícil de comprender.

Ejemplo Composite

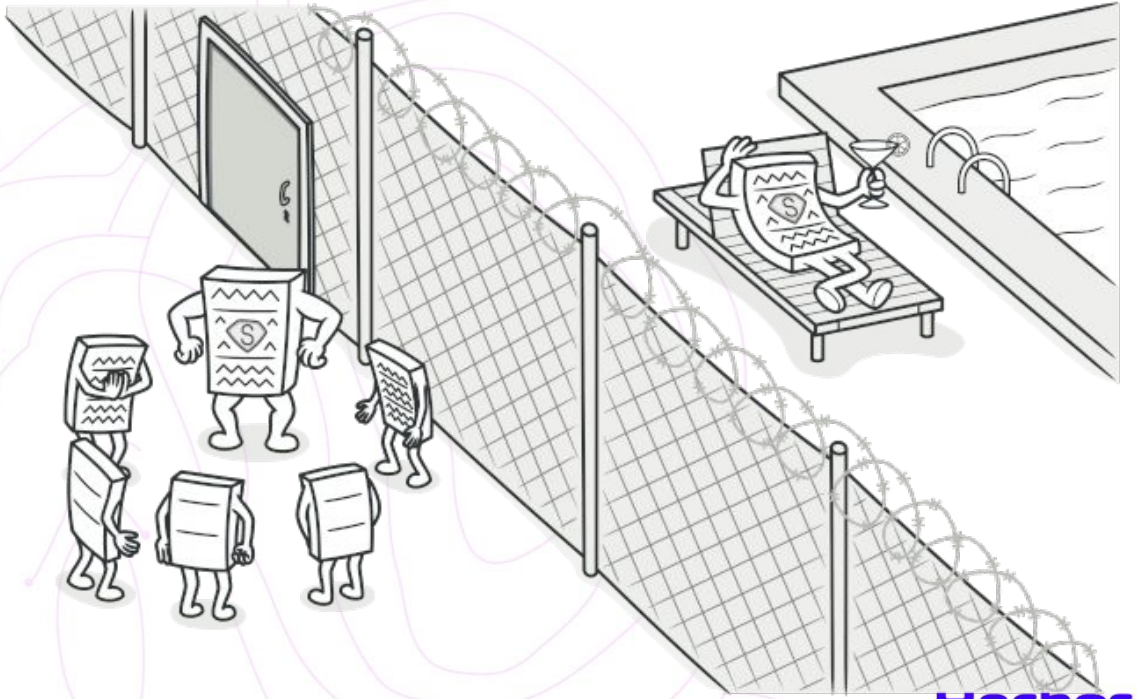


Proxy

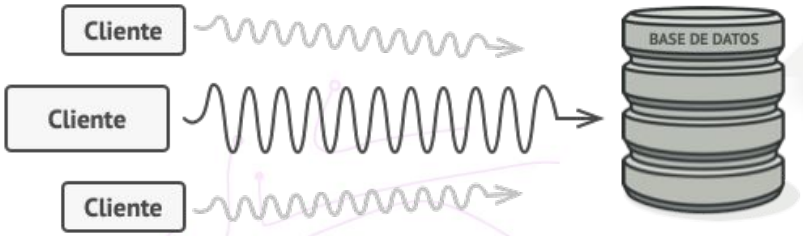
Propósito

Es un patrón que te permite proporcionar un sustituto o marcador de posición para otro objeto.

Un proxy controla el acceso al objeto original, permitiendo hacer algo antes o después de que la solicitud llegue al objeto original.



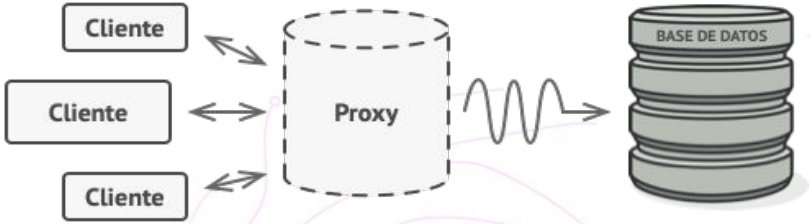
Proxy



Problema

- Imagina que tienes un objeto enorme que consume una gran cantidad de recursos del sistema. Lo necesitas de vez en cuando, pero no siempre.
- Puedes llevar a cabo una implementación diferida, es decir, crear este objeto sólo cuando sea realmente necesario.
- Todos los clientes del objeto tendrán que ejecutar algún código de inicialización diferida. Lamentablemente, esto seguramente generará una gran cantidad de código duplicado.
- En un mundo ideal, querríamos meter este código directamente dentro de la clase de nuestro objeto, pero eso no siempre es posible. Por ejemplo, la clase puede ser parte de una biblioteca cerrada de un tercero.

Proxy

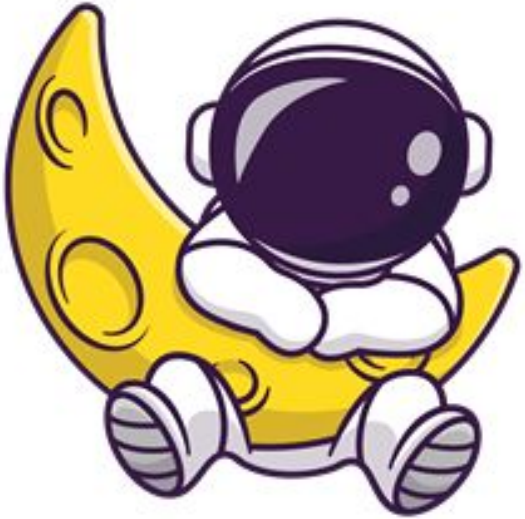


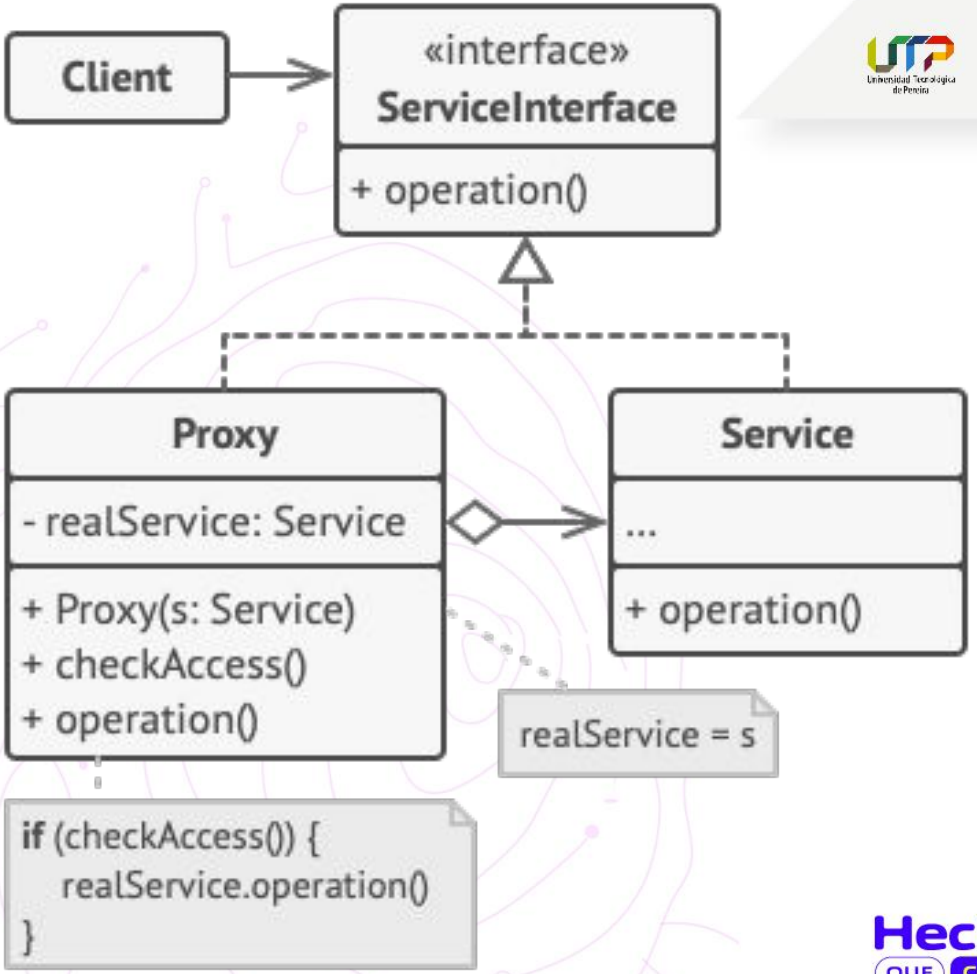
Solución

- El patrón sugiere que crees una nueva clase proxy con la misma interfaz que un objeto de servicio original.
- Después actualizas tu aplicación para que pase el objeto proxy a todos los clientes del objeto original. Al recibir una solicitud de un cliente, el proxy crea un objeto de servicio real y le delega todo el trabajo.
- Si necesitas ejecutar algo antes o después de la lógica primaria de la clase, el proxy te permite hacerlo sin cambiar esa clase. Ya que el proxy implementa la misma interfaz que la clase original, puede pasarse a cualquier cliente que espere un objeto de servicio real.

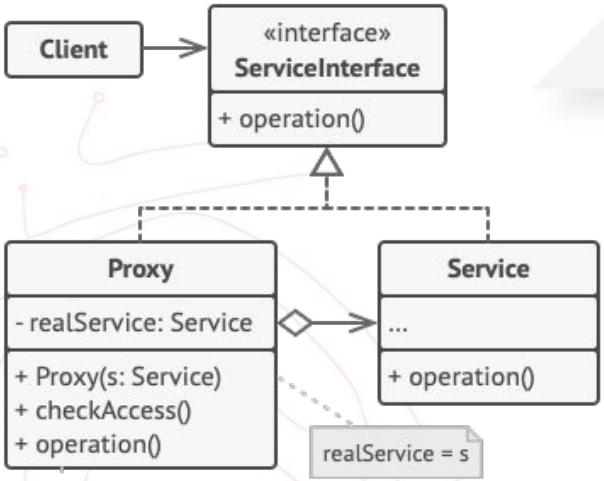
Proxy

Solución General





Proxy



Ventajas

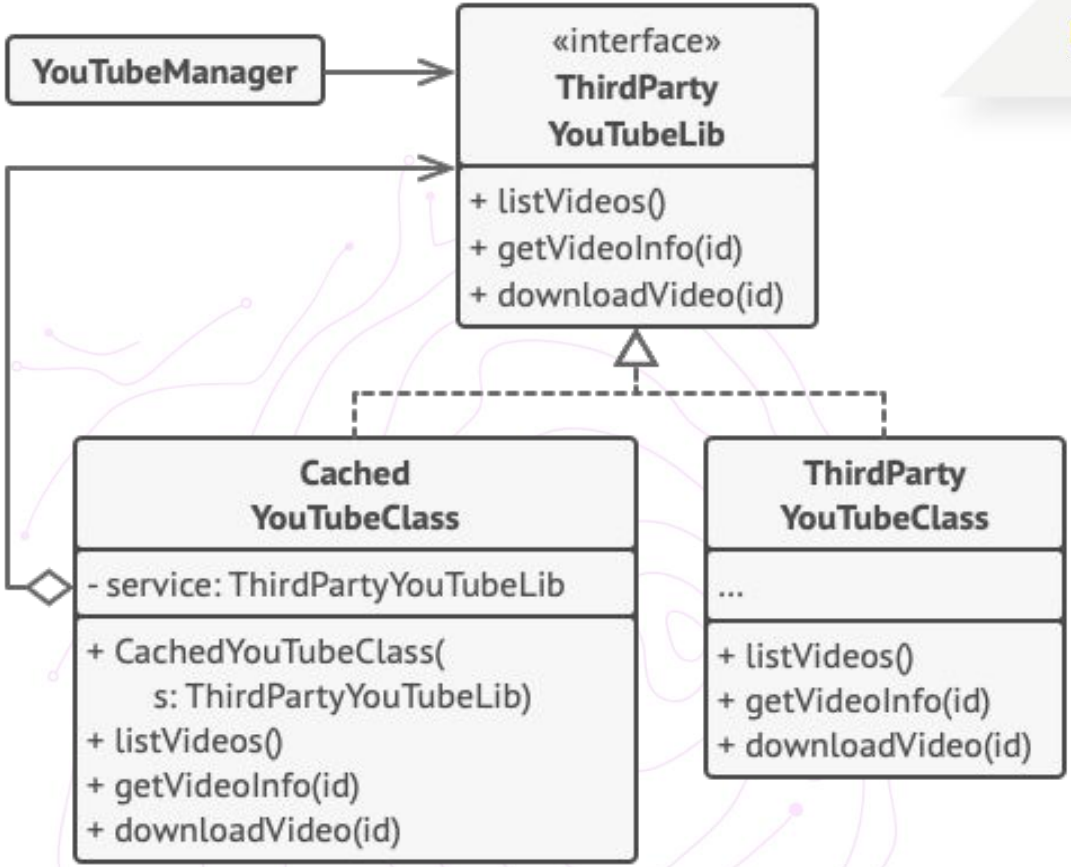
- Puedes controlar el objeto de servicio sin que los clientes lo sepan.
- Puedes gestionar el ciclo de vida del objeto de servicio cuando a los clientes no les importa.
- El proxy funciona incluso si el objeto de servicio no está listo o no está disponible.
- ***Principio de abierto/cerrado***. Puedes introducir nuevos proxies sin cambiar el servicio o los clientes.

Desventajas

- El código puede complicarse ya que debes introducir gran cantidad de clases nuevas.
- La respuesta del servicio puede retrasarse.

Proxy

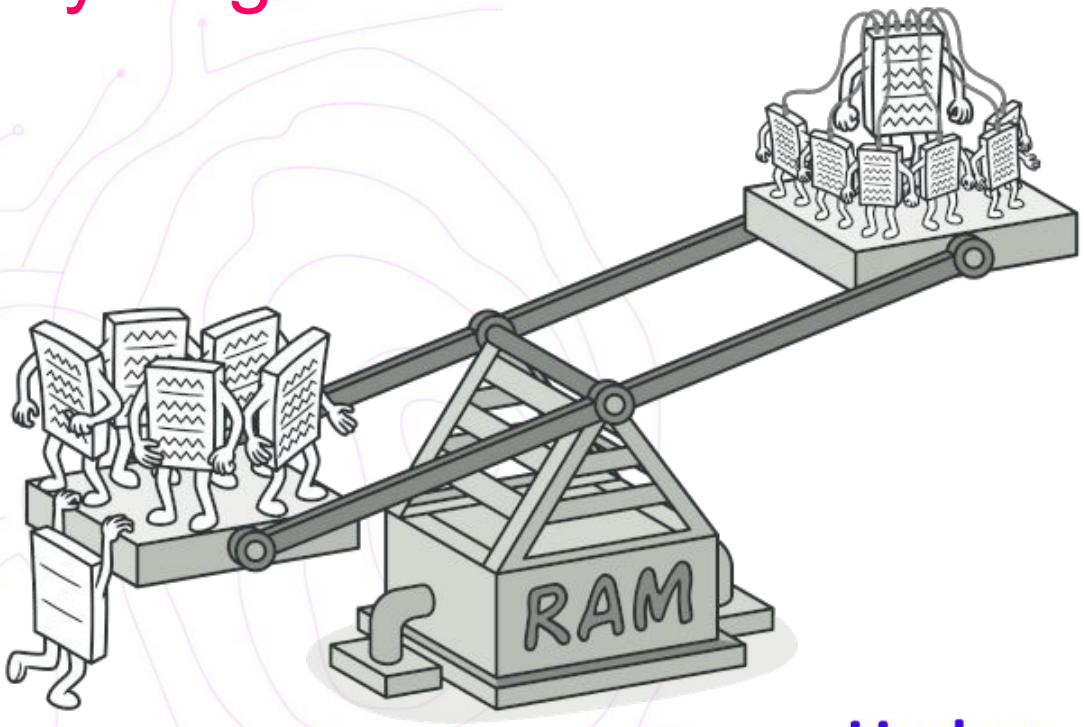
Ejemplo



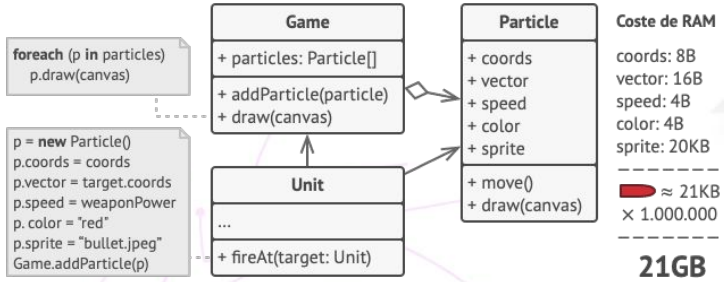
Flyweight

Propósito

Es un patrón que te permite mantener más objetos dentro de la cantidad disponible de RAM compartiendo las partes comunes del estado entre varios objetos en lugar de mantener toda la información en cada objeto.



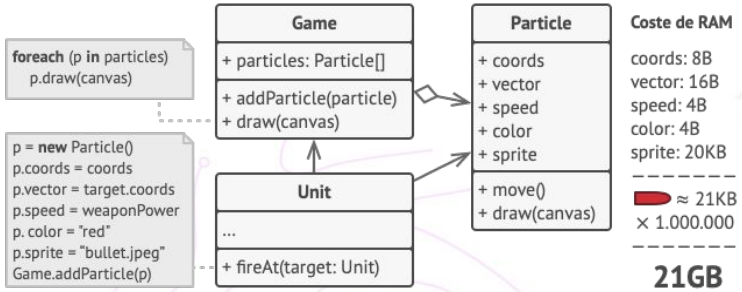
Flyweight



Problema

- Para divertirte un poco después de largas horas de trabajo, decides crear un sencillo videojuego en el que los jugadores se tienen que mover por un mapa disparándose entre sí.
- Decides implementar un sistema de partículas realistas que lo distinga de otros juegos. Grandes cantidades de balas, misiles y metralla de las explosiones volarán por todo el mapa, ofreciendo una apasionante experiencia al jugador.
- Al terminarlo, subes el último cambio, compilas el juego y se lo envias a un amigo para una partida de prueba.
- Aunque el juego funcionaba sin problemas en tu máquina, tu amigo no logró jugar durante mucho tiempo. En su computadora el juego se paraba a los pocos minutos de empezar.
- Tras dedicar varias horas a revisar los registros de depuración, descubres que el juego se paraba debido a una cantidad insuficiente de RAM.

Flyweight



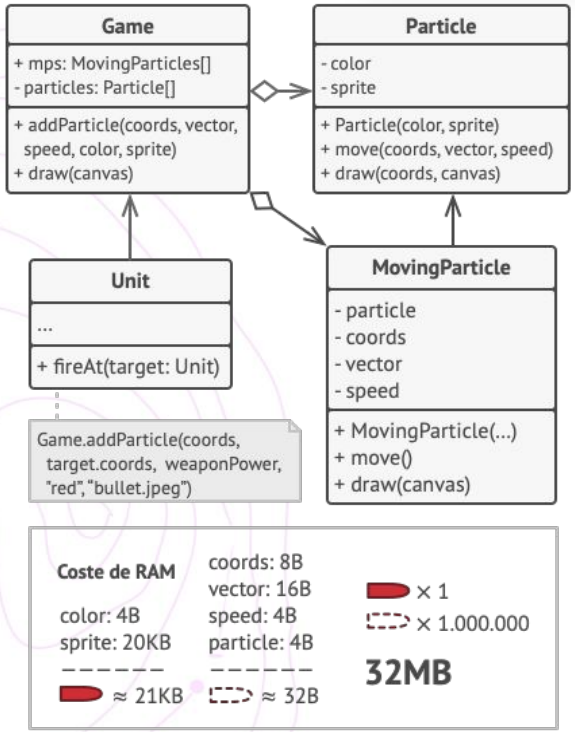
Problema

- Resulta que el equipo de tu amigo es mucho menos potente que tu computadora, y esa es la razón por la que el problema surgió tan rápido en su máquina.
- El problema estaba relacionado con tu sistema de partículas. Cada partícula, como una bala, un misil o un trozo de metralla, estaba representada por un objeto separado que contenía gran cantidad de datos.
- En cierto momento, cuando la masacre alcanzaba su punto culminante en la pantalla del jugador, las partículas recién creadas ya no cabían en el resto de RAM, provocando que el programa fallará.

Flyweight

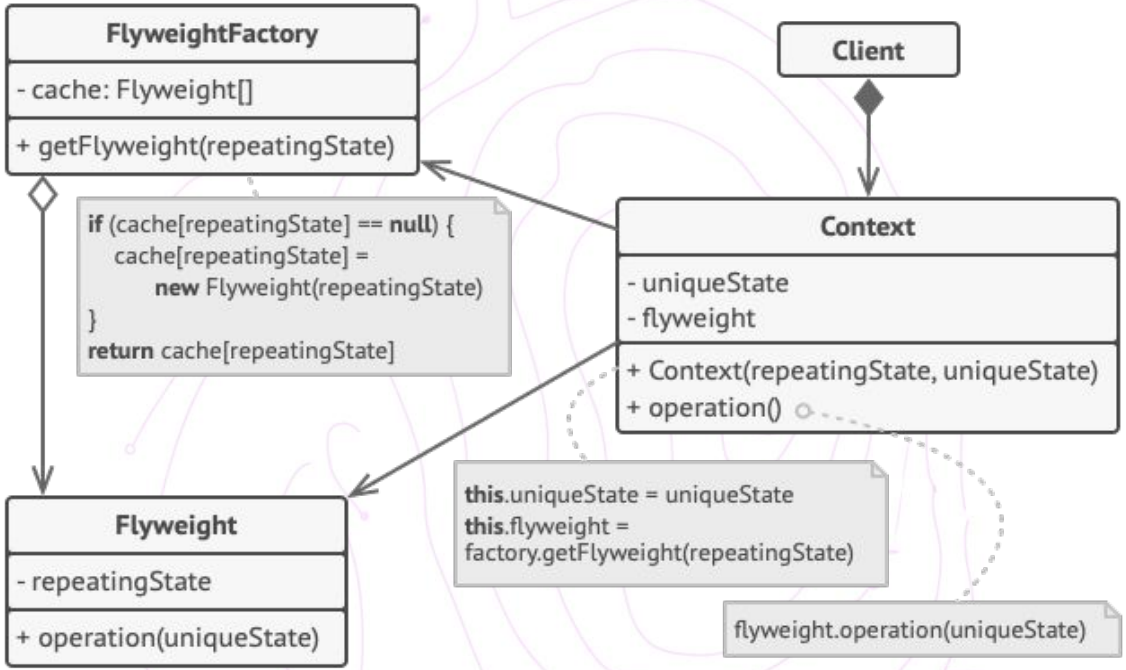
Solución

- Observando más atentamente la clase Partícula, puede ser que te hayas dado cuenta de que los campos de color y sprite consumen mucha más memoria que otros campos.
- Esta información constante de un objeto suele denominarse su estado ***intrínseco***. Existe dentro del objeto y otros objetos únicamente pueden leerla, no cambiarla.
- El resto del estado del objeto, a menudo alterado “desde el exterior” por otros objetos, se denomina el estado ***extrínseco***.



Flyweight

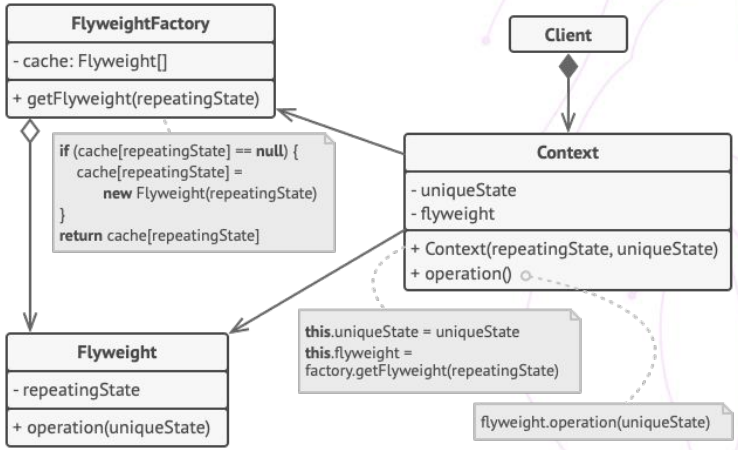
Solución General



Flyweight

Ventajas

- Puedes ahorrar mucha RAM, siempre que tu programa tenga toneladas de objetos similares.

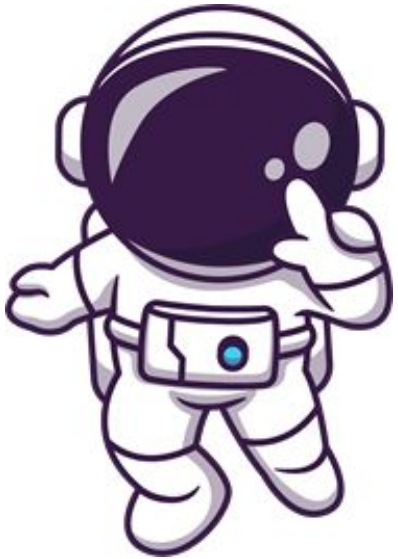


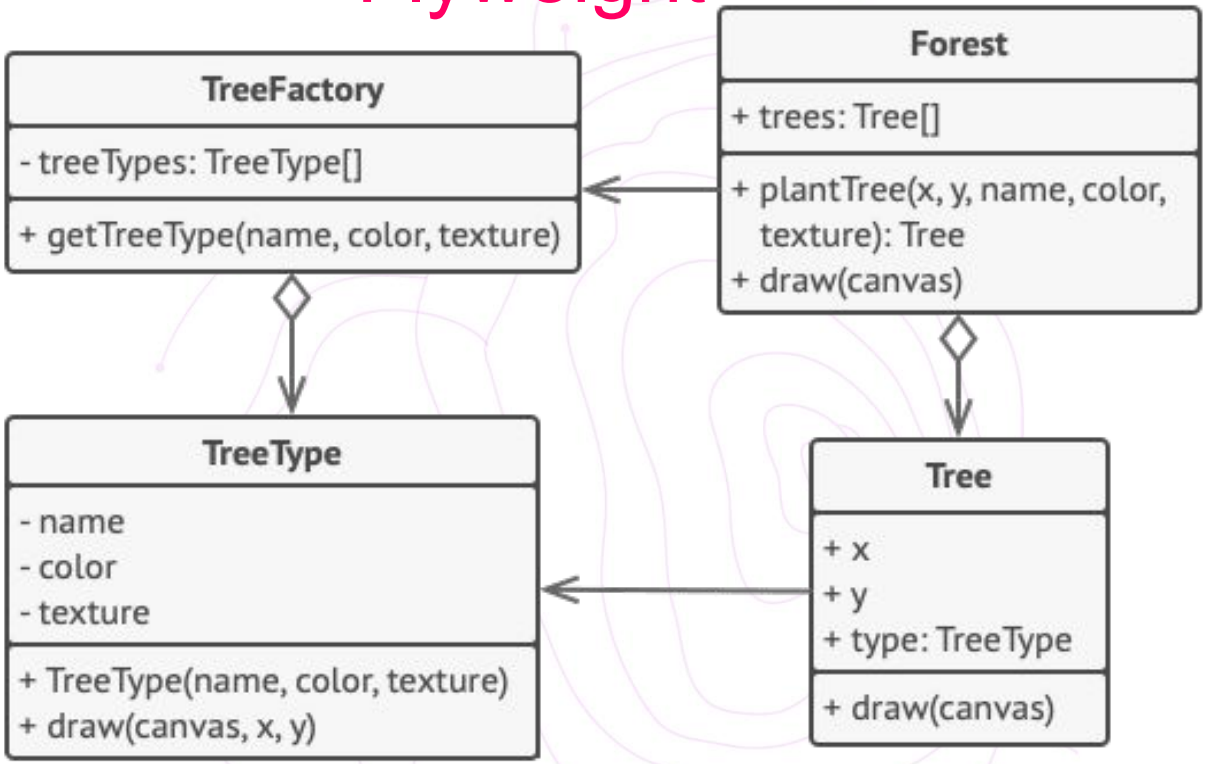
Desventajas

- Puede que estés cambiando RAM por ciclos CPU cuando deba calcularse de nuevo parte de la información de contexto cada vez que alguien invoque un método flyweight.
- El código se complica mucho. Los nuevos miembros del equipo siempre estarán preguntándose por qué el estado de una entidad se separó de tal manera.

Flyweight

Ejemplo





Universidad Tecnológica de Pereira

Mision TIC 2022

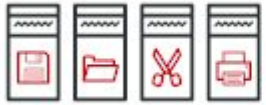
El futuro digital es de todos

MinTIC

Patrones de comportamiento

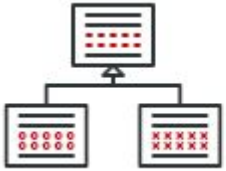
Hechos QUE CONECTAN

Chain of Responsibility

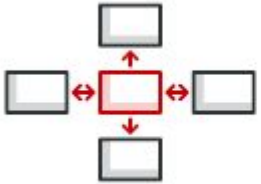


Command

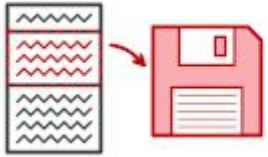
Iterator



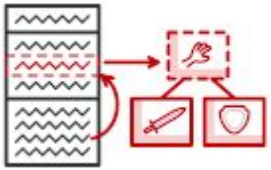
Template Method



Mediator

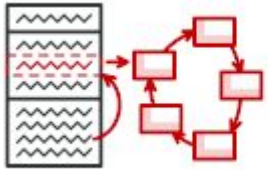


Memento

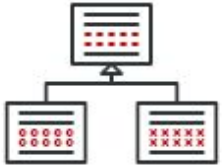


Strategy

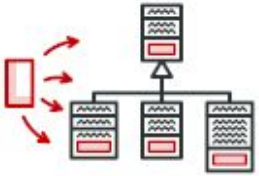
Observer



State



Template Method



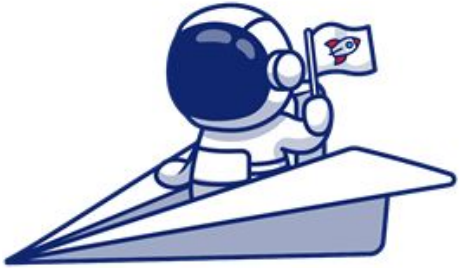
Visitor

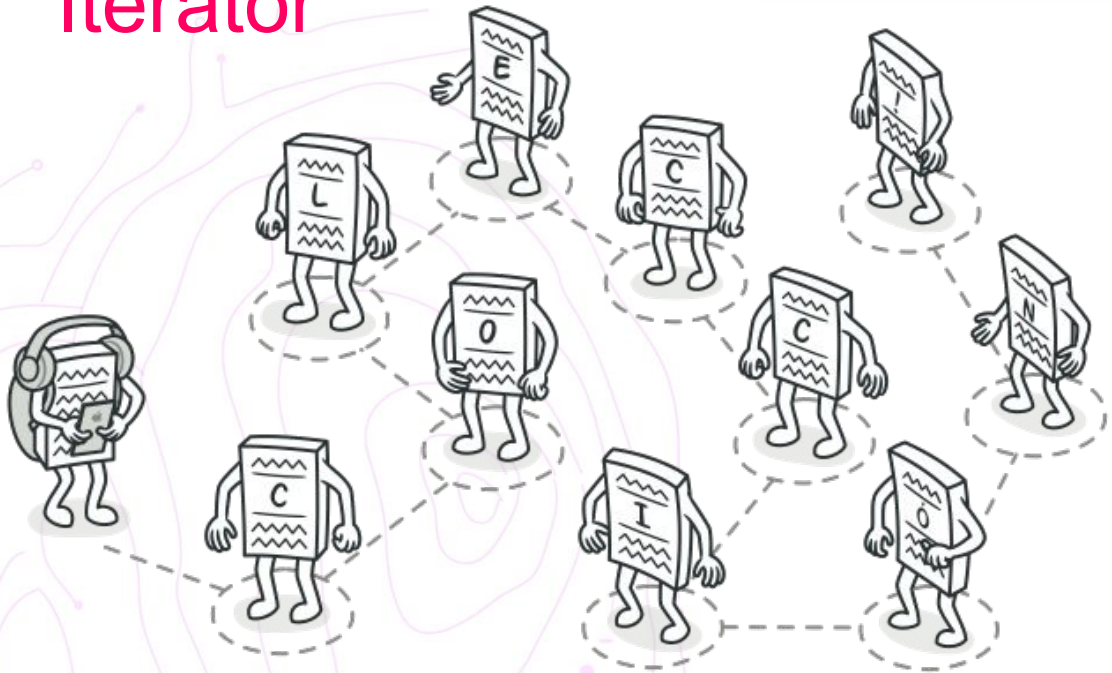
Patrones de comportamiento

Iterator

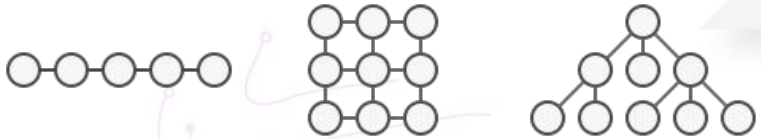
Propósito

Es un patrón que te permite recorrer elementos de una colección sin exponer su representación subyacente (lista, pila, árbol, etc.).





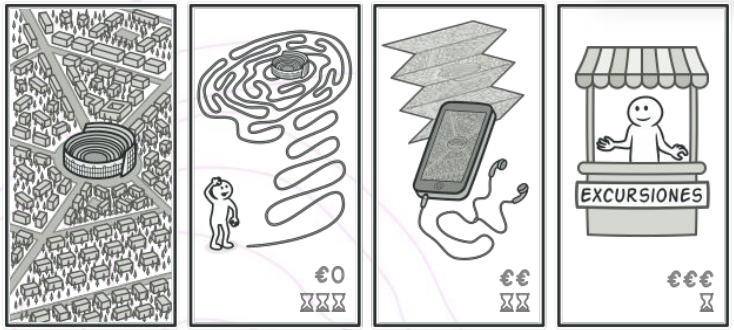
Iterator



Problema

- Las colecciones son de los tipos de datos más utilizados en programación. Sin embargo, una colección tan solo es un contenedor para un grupo de objetos.
- La mayoría de las colecciones almacena sus elementos en simples listas, pero algunas de ellas se basan en pilas, árboles, grafos y otras estructuras complejas de datos.
- Independientemente de cómo se estructure una colección, debe aportar una forma de acceder a sus elementos de modo que otro código pueda utilizar dichos elementos. Debe haber una forma de recorrer cada elemento de la colección sin acceder a los mismos elementos una y otra vez.

Iterator

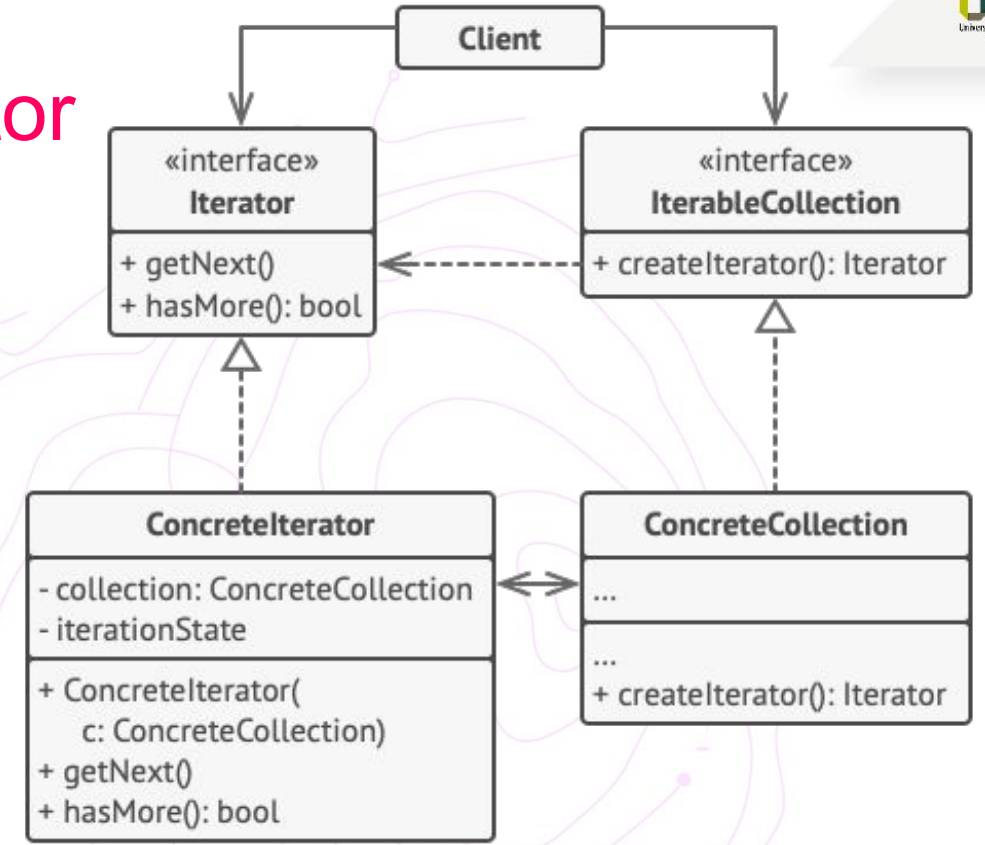


Solución

- La idea central del patrón Iterator es extraer el comportamiento de recorrido de una colección y colocarlo en un objeto independiente llamado iterador.
- Además de implementar el propio algoritmo, un objeto iterador encapsula todos los detalles del recorrido, como la posición actual y cuántos elementos quedan hasta el final. Debido a esto, varios iteradores pueden recorrer la misma colección al mismo tiempo, independientemente los unos de los otros.
- Normalmente, los iteradores aportan un método principal para extraer elementos de la colección. El cliente puede continuar ejecutando este método hasta que no devuelva nada, lo que significa que el iterador ha recorrido todos los elementos.

Iterator

Solución General



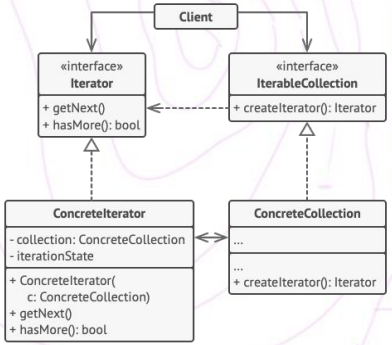
Iterator

Ventajas

- ***Principio de responsabilidad única***. Puedes limpiar el código cliente y las colecciones extrayendo algoritmos de recorrido voluminosos y colocándolos en clases independientes.
- ***Principio de abierto/cerrado***. Puedes implementar nuevos tipos de colecciones e iteradores y pasarlos al código existente sin descomponer nada.
- Puedes recorrer la misma colección en paralelo porque cada objeto iterador contiene su propio estado de iteración.

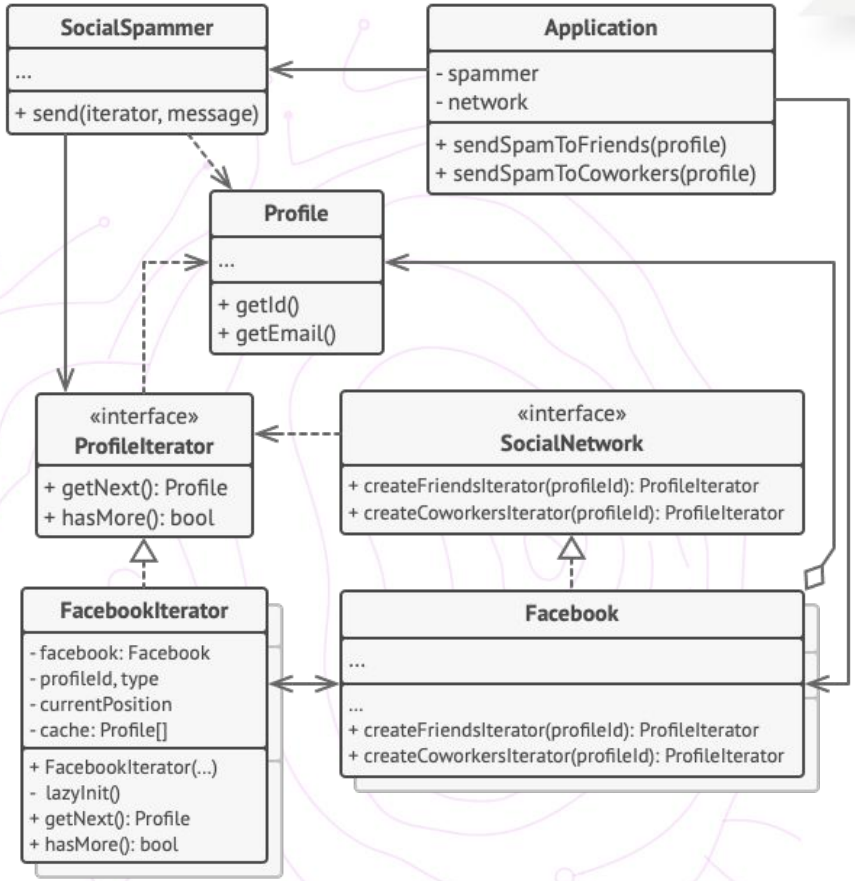
Desventajas

- Aplicar el patrón puede resultar excesivo si tu aplicación funciona únicamente con colecciones sencillas.
- Utilizar un iterador puede ser menos eficiente que recorrer directamente los elementos de algunas colecciones especializadas.



Iterator

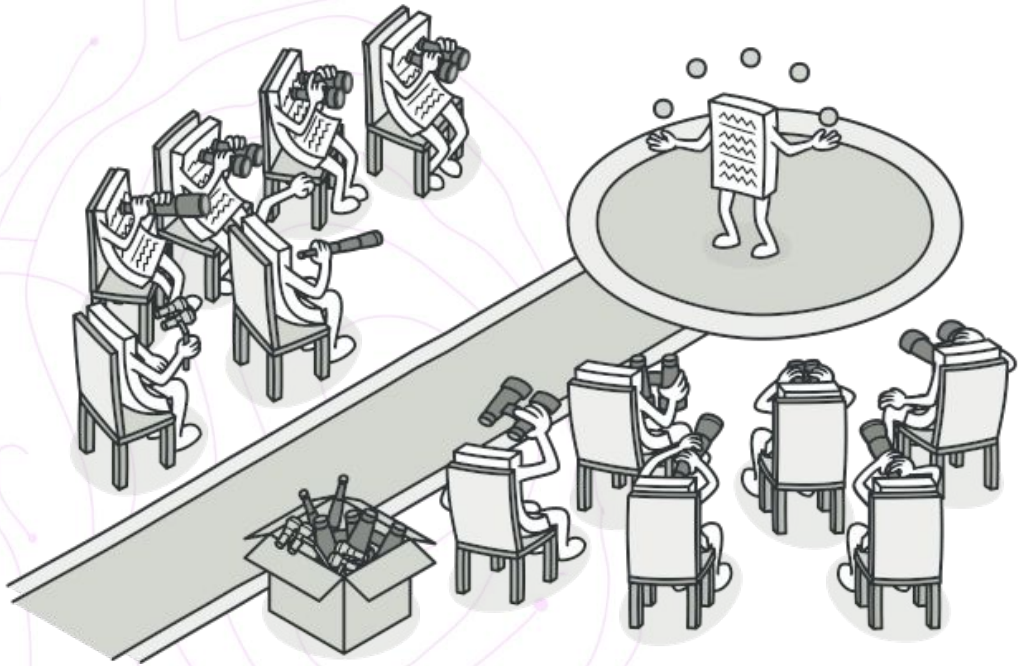
Ejemplo



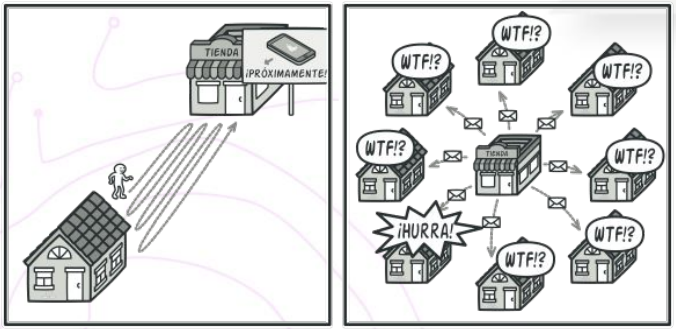
Observer

Propósito

Es un patrón que te permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.



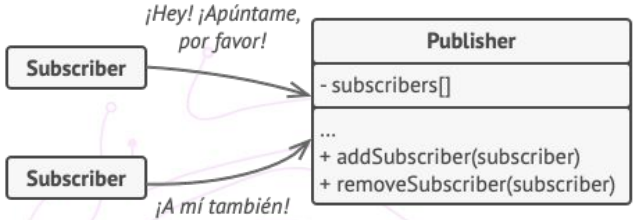
Observer



Problema

- Imagina que tienes dos tipos de objetos: un objeto *Cliente* y un objeto *Tienda*.
- El cliente está muy interesado en una marca particular de producto (digamos, un nuevo modelo de iPhone) que estará disponible en la tienda muy pronto. El cliente puede visitar la tienda cada día para comprobar la disponibilidad del producto. Pero, mientras el producto está en camino, la mayoría de estos viajes serán en vano.
- Parece que nos encontramos ante un conflicto. O el cliente pierde tiempo comprobando la disponibilidad del producto, o bien la tienda desperdicia recursos notificando a los clientes equivocados.

Observer

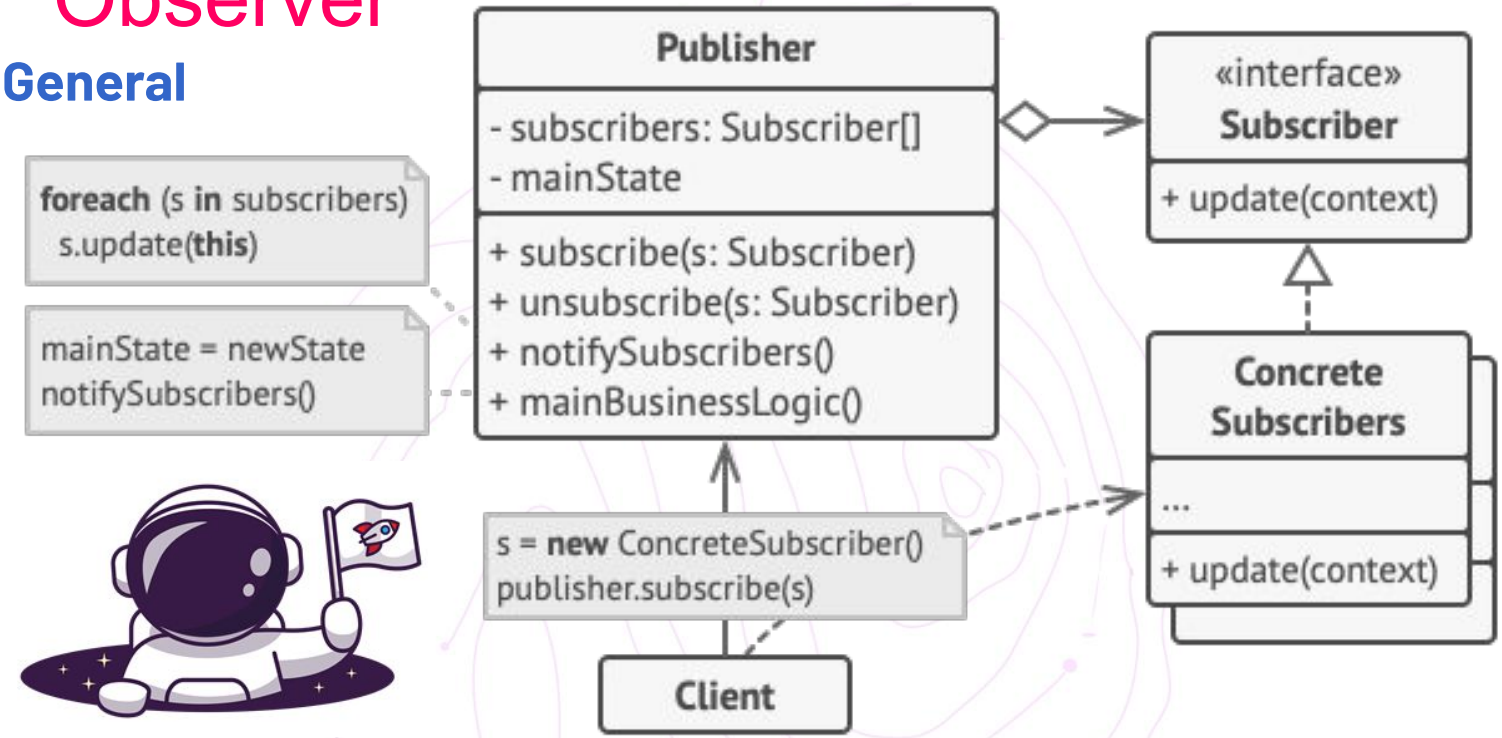


Solución

- El objeto que tiene un estado interesante suele denominarse *sujeto*, pero, como también va a notificar a otros objetos los cambios en su estado, le llamaremos *notificador* (en ocasiones también llamado *publicador*). El resto de los objetos que quieren conocer los cambios en el estado del notificador, se denominan *suscriptores*.
- El patrón Observer sugiere que añadas un mecanismo de suscripción a la clase notificadora para que los objetos individuales puedan suscribirse o cancelar su suscripción a un flujo de eventos que proviene de esa notificadora.
- Este mecanismo consiste en:
  a. ***Un campo matriz para almacenar una lista de referencias a objetos suscriptores.***
  b. ***Varios métodos que permiten añadir suscriptores y eliminarlos de esa lista.***

Observer

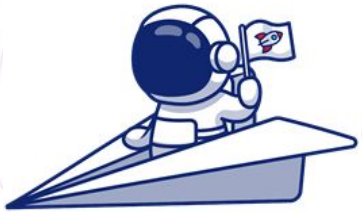
Solución General



Observer

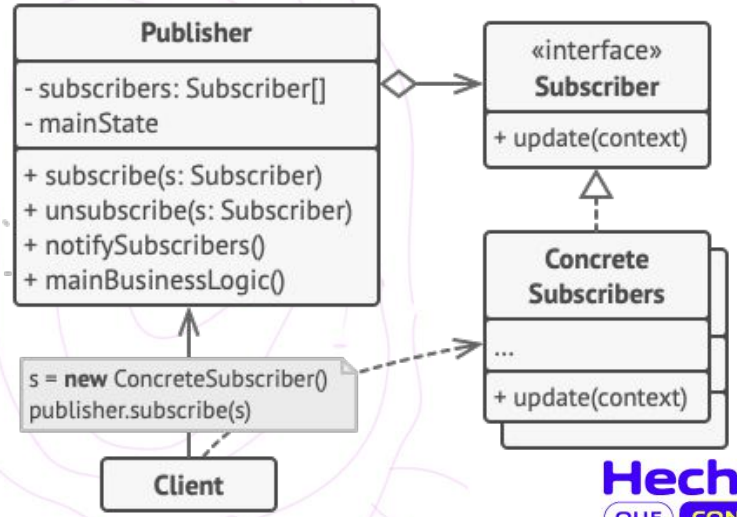
Ventajas

- ***Principio de abierto/cerrado***. Puedes introducir nuevas clases suscriptoras sin tener que cambiar el código de la notificadora (y viceversa si hay una interfaz notificadora).
- Puedes establecer relaciones entre objetos durante el tiempo de ejecución.



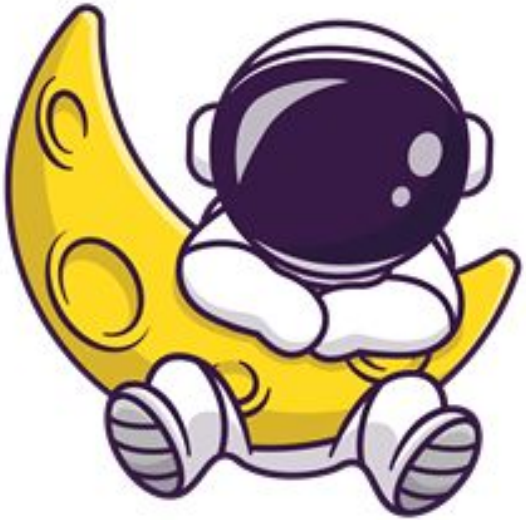
Desventajas

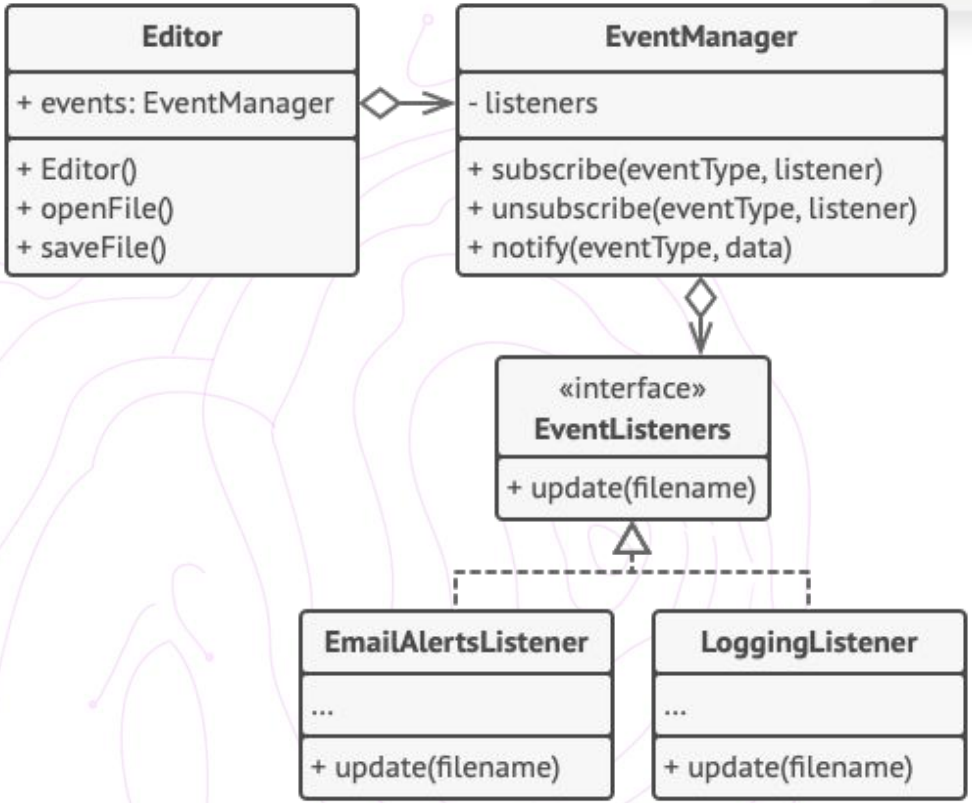
- Los suscriptores son notificados en un orden aleatorio.



Observer

Ejemplo

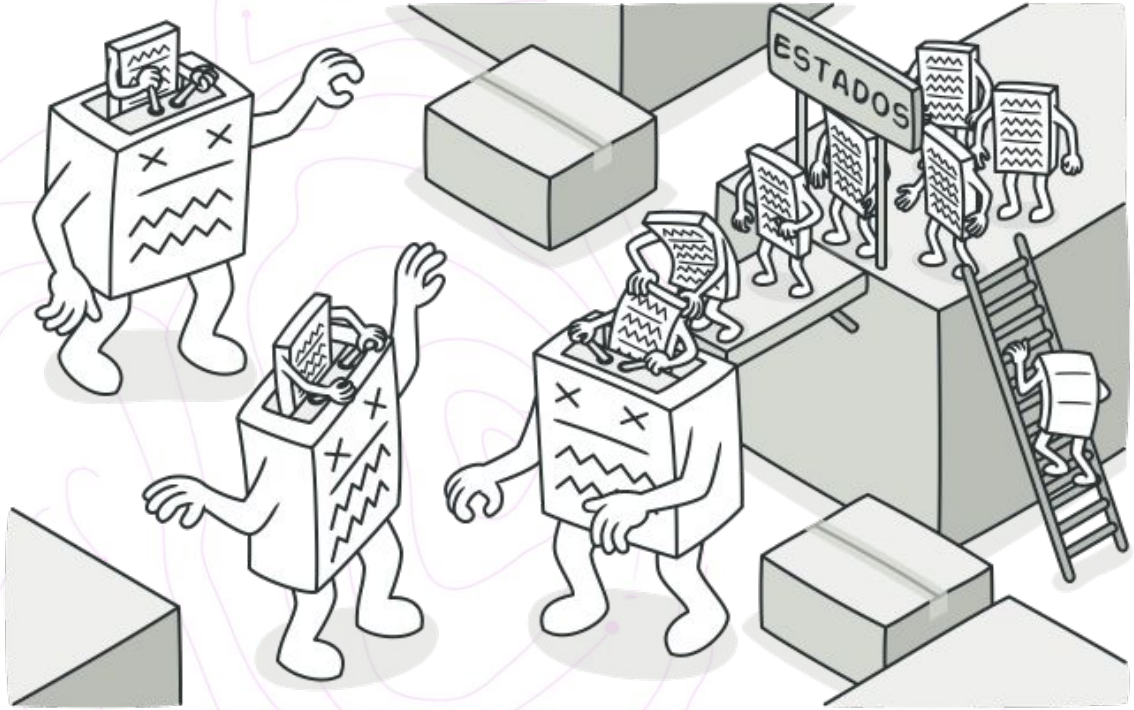




State / Estado

Propósito

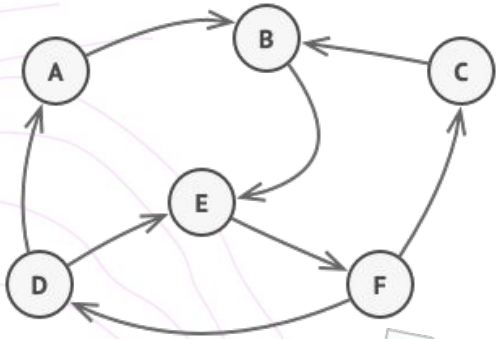
Es un patrón que permite a un objeto alterar su comportamiento cuando su estado interno cambia. Parece como si el objeto cambiara su clase.

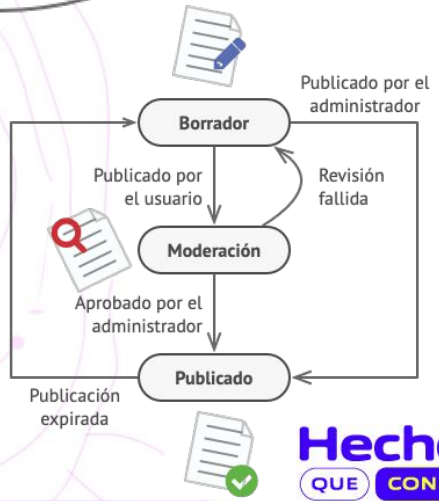


State

Problema

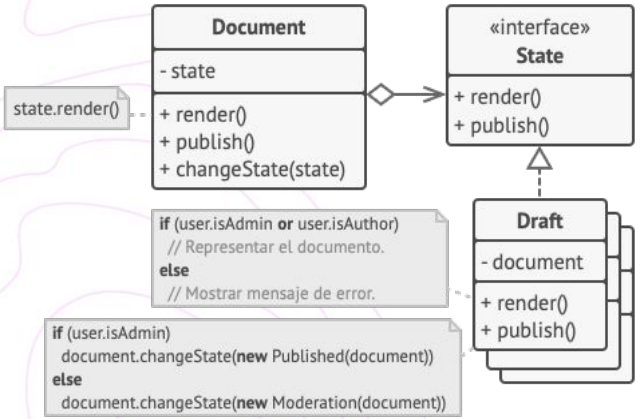
- La idea principal es que, en cualquier momento dado, un programa puede encontrarse en un número finito de estados.
- Dentro de cada estado único, el programa se comporta de forma diferente y puede cambiar de un estado a otro instantáneamente. Sin embargo, dependiendo de un estado actual, el programa puede cambiar o no a otros estados.
- Estas normas de cambio llamadas transiciones también son finitas y predeterminadas.





State

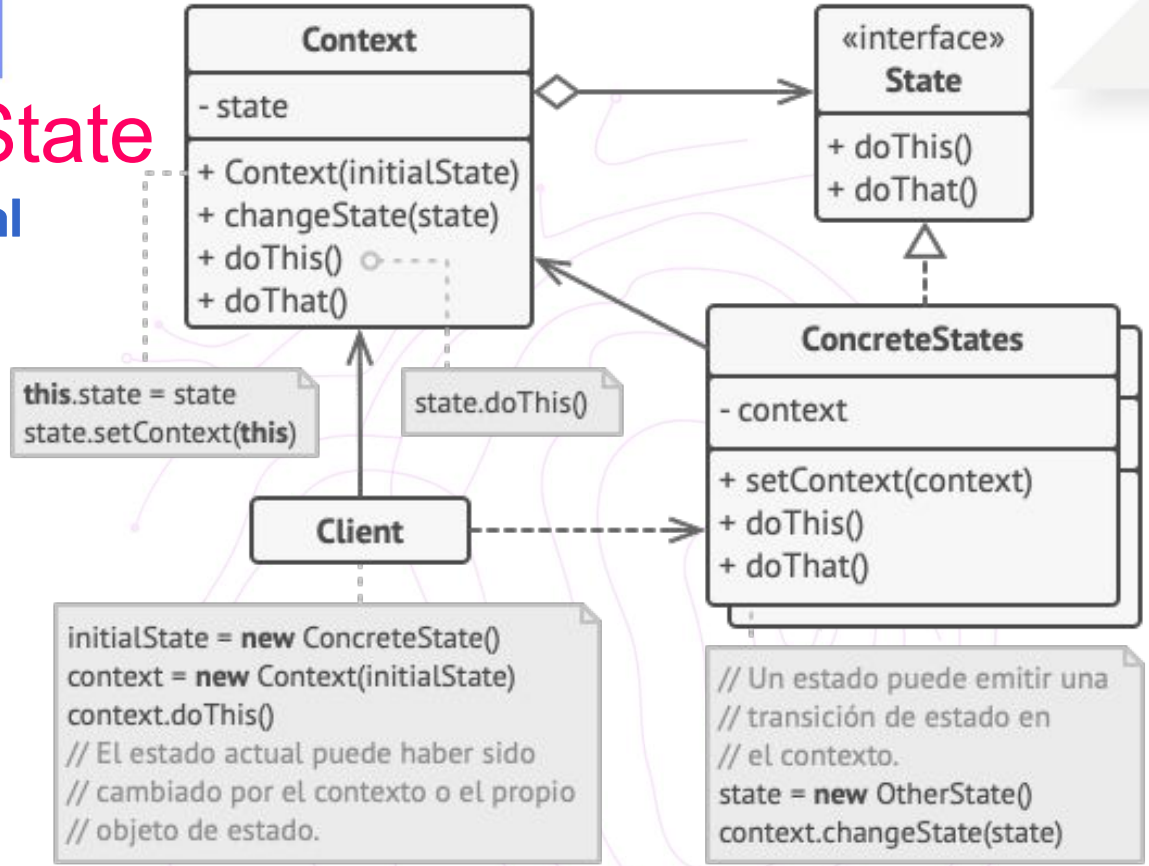
Solución



- El patrón sugiere que crees nuevas clases para todos los estados posibles de un objeto y extraigas todos los comportamientos específicos del estado para colocarlos dentro de esas clases.
- En lugar de implementar todos los comportamientos por su cuenta, el objeto original, llamado *contexto*, almacena una referencia a uno de los objetos de estado que representa su estado actual y delega todo el trabajo relacionado con el estado a ese objeto.

State

Solución General



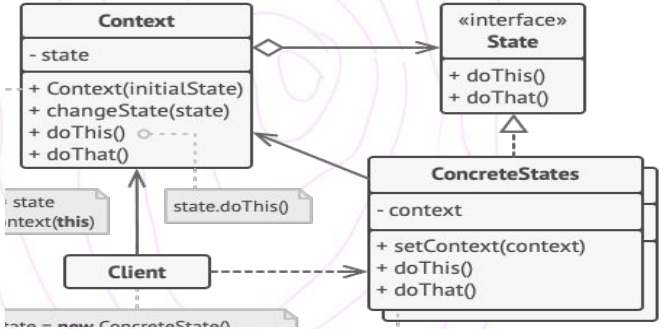
State

Ventajas

- ***Principio de responsabilidad única***. Organiza el código relacionado con estados particulares en clases separadas.
- ***Principio de abierto/cerrado***. Introduce nuevos estados sin cambiar clases de estado existentes o la clase contexto.
- Simplifica el código del contexto eliminando voluminosos condicionales de máquina de estados.

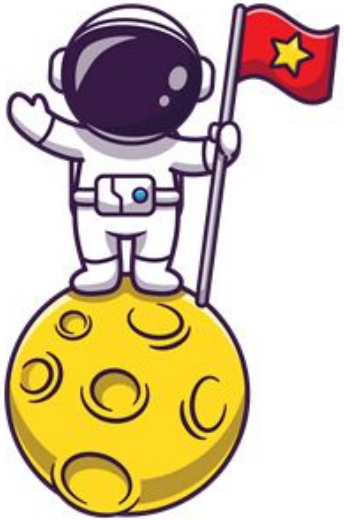
Desventajas

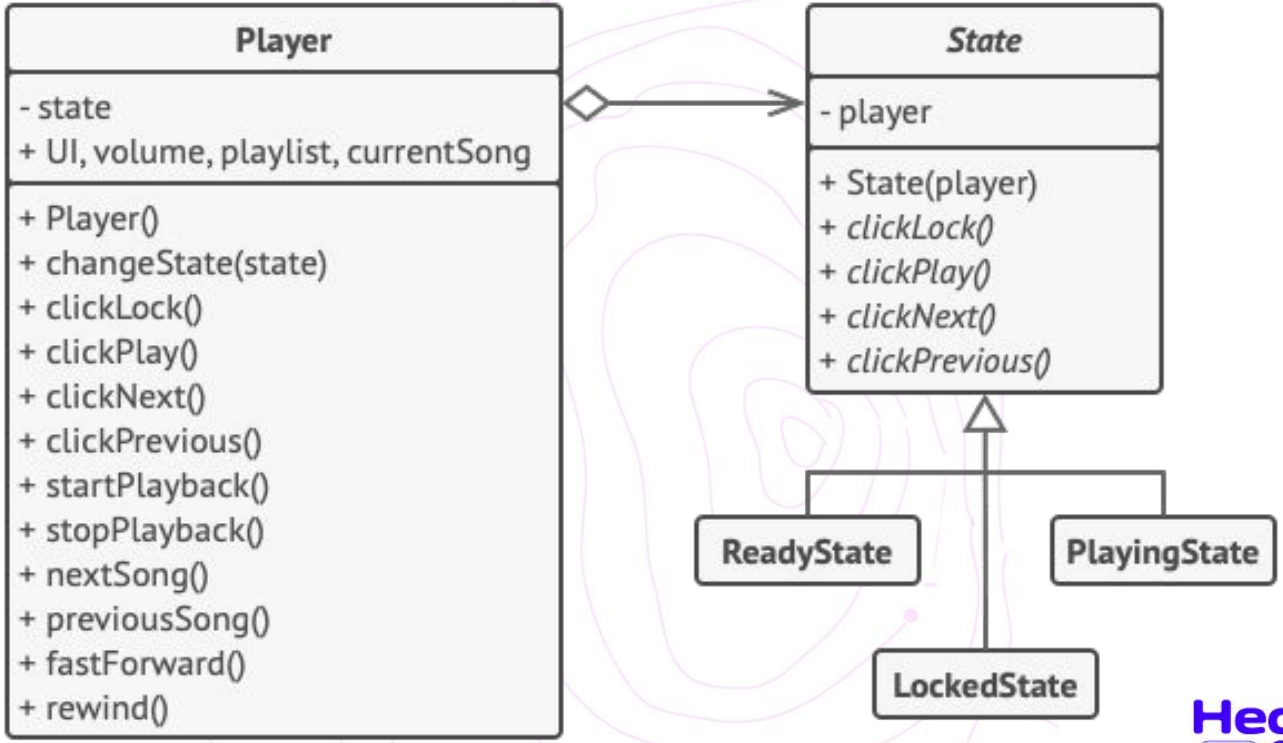
- Aplicar el patrón puede resultar excesivo si una máquina de estados sólo tiene unos pocos estados o raramente cambia.



State

Ejemplo



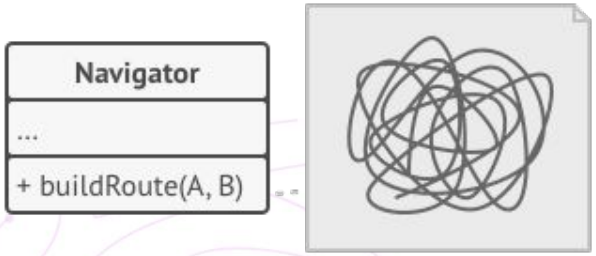


Strategy / Estrategia

Propósito

Es un patrón que te permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.

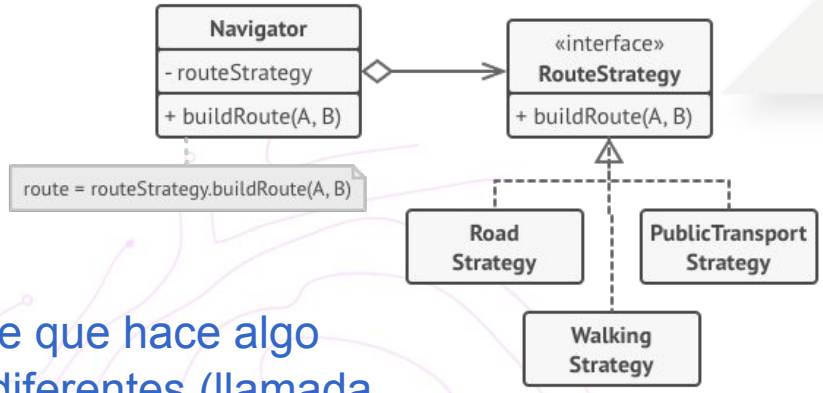
Strategy



Problema

- Un día decidiste crear una aplicación de navegación para viajeros ocasionales. La aplicación giraba alrededor de un bonito mapa que ayudaba a los usuarios a orientarse rápidamente en cualquier ciudad.Una de las funciones más solicitadas para la aplicación era la planificación automática de rutas.
- ***La primera versión*** de la aplicación sólo podía generar las rutas sobre carreteras para las personas en coche.
- ***La siguiente actualización***, se agrega una opción para crear rutas a pie.
- ***Después***, se agrega otra opción para permitir a las personas utilizar el transporte público en sus rutas.
- ***Más tarde*** se planea añadir la generación de rutas para ciclistas, y ***más tarde***, otra opción para trazar rutas por todas las atracciones turísticas de una ciudad.

Strategy

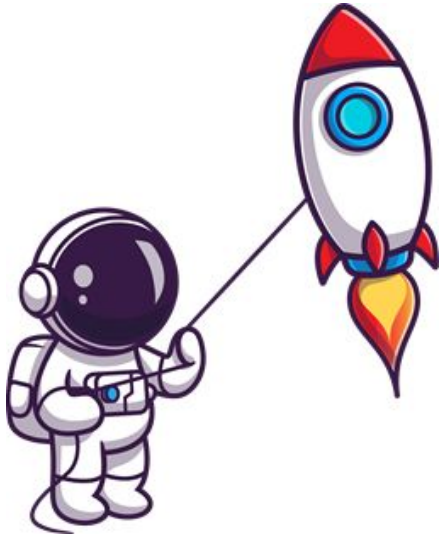


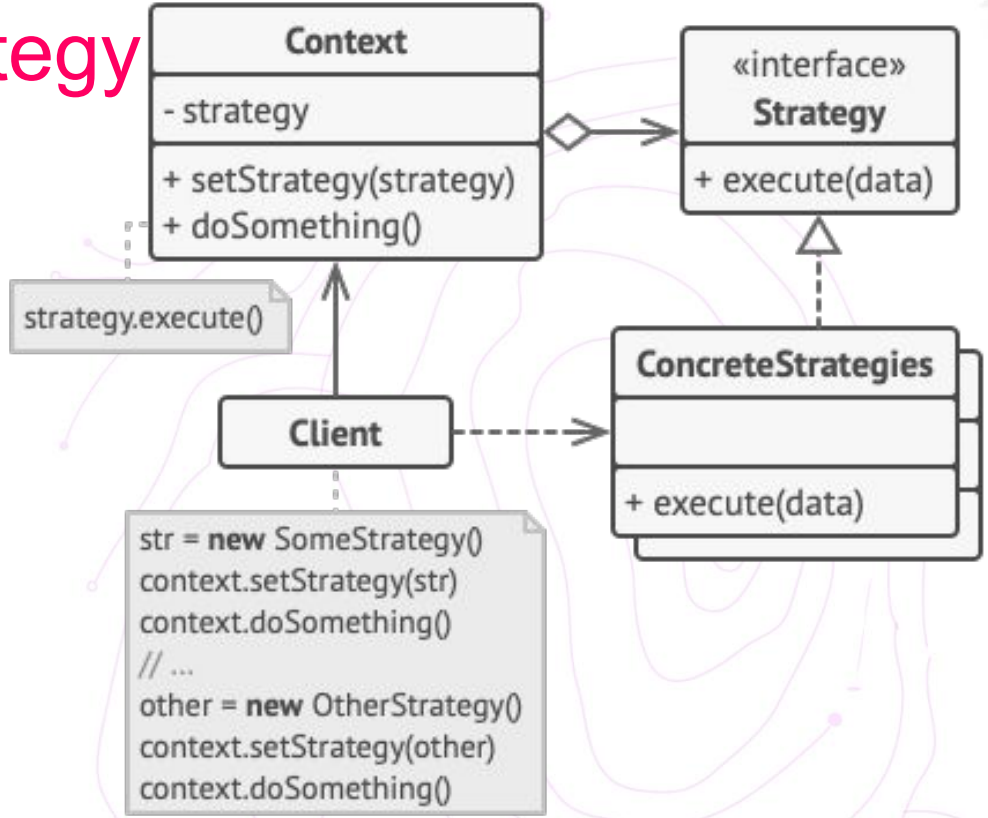
Solución

- El patrón sugiere tomar la clase que hace algo específico de muchas formas diferentes (llamada ***contexto***) y extraiga todos esos algoritmos para colocarlos en clases separadas llamadas ***estrategias***.
- La clase contexto no es responsable de seleccionar un algoritmo adecuado para la tarea. En lugar de eso, el cliente pasa la estrategia deseada a la clase contexto. De hecho, la clase contexto no sabe mucho acerca de las estrategias. Funciona con todas las estrategias a través de la misma interfaz genérica, que sólo expone un único método para disparar el algoritmo encapsulado dentro de la estrategia seleccionada.

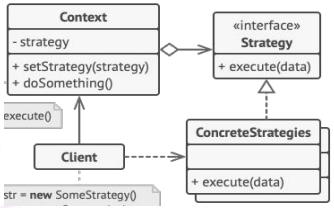
Strategy

Solución General





Strategy



Ventajas

- Puedes intercambiar algoritmos usados dentro de un objeto durante el tiempo de ejecución.
- Puedes aislar los detalles de implementación de un algoritmo del código que lo utiliza.
- Puedes sustituir la herencia por composición.
- ***Principio de abierto/cerrado***. Puedes introducir nuevas estrategias sin tener que cambiar el contexto.

Desventajas

- Si sólo tienes un par de algoritmos que raramente cambian, no hay una razón real para complicar el programa.
- Los clientes deben conocer las diferencias entre estrategias para poder seleccionar la adecuada.
- Muchos lenguajes de programación modernos tienen un soporte de tipo funcional que te permite implementar distintas versiones de un algoritmo dentro de un grupo de funciones anónimas.